

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 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 четыре академических часа.

Каждая лабораторная работа состоит из теоретического материала, заданий, обязательных для выполнения, дополнительных заданий, а также контрольных вопросов для проверки усвоения теоретического материала.

При оформлении отчетов по лабораторным работам необходимо использовать приложение Word. Каждый отчет должен содержать название работы, ответы на контрольные вопросы, условия задач, 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. Представление информации в виде структуры. Индексирование. Union. Enum	4
Лабораторная работа № 2. Создание меню программы. Разбиение программы на файлы. STL контейнеры. Array	22
Лабораторная работа № 3. Строки типа string. Фильтрация данных. Vector	29
Лабораторная работа № 4 Двоичные и текстовые файлы. Запись/чтение массива структур в файл. Deque	41
Лабораторная работа № 5. Редактирование файлов: частичное и полное удаление, изменение поля. Редактирование массива структур. Forward_list	57
Лабораторная работа № 6. Сортировки. List	68
Лабораторная работа № 7. Методы поиска. Set	86
Лабораторная работа № 8. Определение сложности алгоритма. Map	97
Приложение 1. Изменение цвета консоли	112
Приложение 2. Многопоточность	114
Приложение 3. Измерение времени выполнения кода	117
Приложение 4. Стандартная библиотека C++ algorithm	119

Лабораторная работа № 1. Представление информации в виде структуры. Индексирование. Union. Enum

Цель: ознакомиться с работой со структурами, массивами структур, создать структуру, на которой будет базироваться будущая работа, рассмотреть структуру как член структуры. Изучить методы работы с union и enum.

Теоретические сведения № 1

Структура — это пользовательский тип данных, совокупность логически связанных данных различного типа, объединенных под одним идентификатором. В языках C или C++ структура создается при помощи ключевого слова struct, за которым следуют необязательное поле тега и список элементов структуры.

```
struct поле_тега {  
    тип_элемента элемент 1;  
    тип_элемента элемент 2;  
    ...  
    тип_элемента_элемент N;  
}; // точка с запятой обязательны
```

Членами структуры могут быть примитивные типы, массивы, другие структуры и т.д.

```
struct Person  
{  
    char firstname[20];  
    char hobby[30];  
    int age;  
};
```

В C++, есть несколько способов получения доступа к членам структуры:

1. Оператор точки (.) - этот оператор используется для доступа к членам структуры через имя экземпляра структуры. Например:

```
#include <iostream>  
using namespace std;  
  
struct Person  
{  
    char firstname[20];  
    char job[30];  
    int age;  
};
```

```

int main() {
    setlocale(LC_CTYPE, "rus");
    Person john;
    strcpy_s(john.firstname, "Джон");
    john.age = 30;
    strcpy_s(john.job, "Программист");
    cout << "Имя: " << john.firstname << endl; // Имя: Джон
    cout << "Возраст: " << john.age << endl; // Возраст: 30
    cout << "Работа: " << john.job << endl; // Работа: Программист

    return 0;
}

```

2. Оператор стрелки (->) - этот оператор используется для доступа к членам структуры через указатель на экземпляр структуры. Например:

```

#include <iostream>
using namespace std;

struct Person {
    char firstname[20];
    int age;
    char job[30];
};

int main() {
    setlocale(LC_CTYPE, "rus");
    Person* john = new Person;
    strcpy_s(john->firstname, "Джон");
    john->age = 30;
    strcpy_s(john->job, "Программист");
    cout << "Имя: " << john->firstname << endl; // Имя: Джон
    cout << "Возраст: " << john->age << endl; // Возраст: 30
    cout << "Работа: " << john->job << endl; // Работа: программист
    delete john;
    return 0;
}

```

Массив структуры - это массив, содержащий элементы типа структуры. То есть, каждый элемент массива является экземпляром структуры. Массивы структур могут быть очень полезными в C++, когда требуется хранить и обрабатывать

большое количество данных, каждое из которых состоит из нескольких связанных элементов. Обратимся к предыдущему примеру:

```
struct Person
{
    char firstname[20];
    char hobby[30];
    int age;
};
```

Здесь Person - это структура, содержащая три поля: firstname, hobby и age. Теперь мы можем создать массив people, содержащий элементы структуры Person, следующим образом:

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

Здесь мы создаем массив people, содержащий три элемента типа Person. Мы можем обращаться к каждому элементу массива, используя оператор индексации [] и записывать значения его полей. Например:

```
strcpy_s(people[0].firstname, "John");
people[0].age = 30;
strcpy_s(people[0].hobby, "Software Engineer");

strcpy_s(people[1].firstname, "Kate");
people[0].age = 24;
strcpy_s(people[0].hobby, "Sport");
```

Массивы структур могут быть полезными при обработке большого количества данных, таких как записи в файле или результаты запроса к базе данных. Они могут быть использованы для хранения и обработки больших объемов информации, что делает их важным элементом в различных программах на C++.

Обратите внимание, что членами структуры могут быть другие структуры.

```
#include <iostream>

using namespace std;

struct Address {
    char street[50];
```

```

        int number;
        char city[50];
    };

    struct Person {
        char name[50];
        int age;
        Address address;
    };

    int main() {
        setlocale(LC_CTYPE, "rus");
        Person p;
        strcpy_s(p.name, "Джон");
        p.age = 30;
        strcpy_s(p.address.street, "Багряный переулок");
        p.address.number = 123;
        strcpy_s(p.address.city, "Минск");

        cout << "Имя: " << p.name << endl; // Имя: Джон
        cout << "Возраст: " << p.age << endl; // Возраст: 30
        cout << "Адрес: " << p.address.number << " " << p.address.street << ", "
        << p.address.city << endl;
        // Адрес: 123 Багряный переулок, Минск
        return 0;
    }

```

Здесь структура Address представляет адрес, а структура Person содержит имя, возраст и адрес человека. Обратите внимание, что для доступа к членам структуры Address из структуры Person используется оператор точки (.).

В данной лабораторной работе перед вами стоит цель создать структуру, которая будет использоваться в дальнейшем при создании полноценной базы данных.

Теоретические сведения № 2

Объединение – это группирование переменных, которые разделяют одну и ту же область памяти. В зависимости от интерпретации осуществляется обращение к той или другой переменной объединения.

Объединение позволяет представить в компактном виде данные, которые могут изменяться. Одни и те же данные могут быть представлены разными способами с помощью объединений.

Для определения объединений применяется ключевое слово `union` и следующий синтаксис:

```
union имя_объединения
{
    тип элемент1;
    тип элемент2;
    .....
    тип элементN;
};
```

Рассмотрим объединения на примере. Создадим простейшее объединение, которое хранит символ и его числовой код из таблицы ASCII:

```
union ascii //наше объединение
{
    int digit; //числово код символа
    char letter; //сам символ
};
```

Конкретный размер выделенной памяти будет зависеть от системы и реализации, но в общем случае это будет выглядеть примерно следующим образом:



После определения объединения мы можем создать его переменную и присвоить ей какое-либо значение. При определении переменной объединения мы ее можем сразу инициализировать, но стоит учитывать, что инициализировать мы можем только один элемент объединения:


```
union ascii code = { 120 } //определение переменной и присвоение ей значения целого числа
```

Используем в нашей программе обращение с помощью элемента точка:

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

union ascii
{
    int digit;
    char letter;
};

int main(void)
{
    union ascii code; //определение переменной
    code.digit = 120; // присвоение значения целого числа
    cout << code.digit << "-" << code.letter << endl; // 120 - x
    code.letter = 87;
    cout << code.digit << "-" << code.letter << endl; // 87 - w
    return 0;
}
```

В первой строке кода мы присваиваем значение 120 полю digit переменной code. Затем мы выводим значение поля digit и поля letter на экран. Поскольку значение поля digit равно 120, то значение поля letter будет соответствовать символу с кодом 120 в таблице ASCII, т.е. символу 'x'.

Затем мы присваиваем значение 87 полю letter переменной code. Затем мы выводим значение поля digit и поля letter на экран. Поскольку значение поля letter равно 87, то значение поля digit будет соответствовать целому числу с кодом 87 в таблице ASCII, т.е. числу 87.

Применительно к объединениям существуют несколько ограничений:

1. Размер конечного объединения не может быть больше размера самого большого поля.
2. В объединении можно использовать только одно поле за раз.
3. Объединение не может находиться в статической памяти.
4. Объединение не может иметь конструктор или деструктор.
5. Доступ к полям объединения должен быть осуществлен через используемое в данный момент поле.
6. Объединение нельзя копировать и передавать его в функцию. Если требуется использование объединения в качестве параметра, лучше передавать

ссылку на объединение.

В языке C++ существует специальный тип объединения, называемый *анонимным объединением*. В анонимном объединении не содержится имени класса и не объявляются никакие объекты. Анонимное объединение просто уведомляет компилятор о том, что его переменные-члены должны иметь одну и ту же область памяти. Анонимные объединения должны содержать только данные, никакие функции-члены не допускаются.

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

union
{
    int digit;
    char letter;
} code1, code2;    // переменные code1, code2

int main(void)
{
    code1.digit = 122;
    code2.digit = 84;

    cout << code1.digit << "-" << code1.letter << endl; // 122 - z
    cout << code2.digit << "-" << code2.letter << endl; // 84 - T

    return 0;
}
```

Теоретические сведения № 3

Перечисление — это список именованных целочисленных констант. Перечисления используются в тех случаях, когда переменные создаваемого типа могут принимать заранее известное конечное множество значений.

Для создания перечисления используется ключевое слово `enum`. Общая форма перечисления имеет следующий вид:

```
enum class имя_перечисления { константа_1, константа_2, ... константа_N
};
```

Создадим перечисление:

```
enum Colors {  
    RED,  
    GREEN,  
    BLUE,  
};
```

Каждой константе сопоставляется некоторое числовое значение. По умолчанию первая константа получает в качестве значения 0, а остальные увеличиваются на единицу.

Пример демонстрирующий присвоение значений в перечислении:

```
#include <iostream>  
using namespace std;  
  
enum Colors {  
    RED, //0  
    GREEN, //1  
    BLUE, //2  
};  
  
int main()  
{  
    Colors currentColor = BLUE;  
    cout << "Number of blue color is : " << currentColor << endl; //Number of blue  
    color is : 2  
    return 0;  
}
```

Мы также можем управлять установкой значений в перечислении. Так, мы можем задать начальное значение для одной константы, тогда у последующих констант значение увеличивается на единицу:

```
#include <iostream>  
using namespace std;  
  
enum Colors {  
    RED = 3,  
    GREEN, //4  
    BLUE, //5  
};
```

```

    int main()
    {
        Colors currentColor = BLUE;
        cout << "Number of blue color is : " << currentColor << endl; //Number of blue
color is : 5
        return 0;
    }

```

Можно назначить каждой константе индивидуальное значение или сочетать этот подход с автоустановкой:

```

#include <iostream>
using namespace std;

enum Colors {
    RED = 2,
    ORANGE, //3
    YELLOW, //4
    GREEN, //5
    BLUE, //6
    VIOLETTE = 1
};

int main()
{
    Colors currentColor = BLUE;
    cout << "Number of blue color is : " << currentColor << endl; //Number of blue
color is : 6
    return 0;
}

```

Задания к лабораторной работе №1

Задание 1.

Реализовать следующие структуры:

Вариант 1. Тема: Гостиница

Предстоит работа с организацией брони номеров клиентами.

Реализовать структуры:

1. База номеров. В базе содержится информация об уровне комфорта номера (стандарт, полулюкс, люкс), номер (например, №325), стоимости и

доступности в данный момент (при заказе номера клиента, он становится недоступным)

2. База клиентов. Содержится информация об имени, фамилии, отчестве клиента и его контактный телефон. Для каждого клиента должна иметься информация о заказанных им номерах с датой заселения.

Вариант 2. Тема: Доставка еды

Предстоит работа с отслеживанием деятельности сервиса доставки

Реализовать структуры:

1. База меню. В базе содержится информация о названии блюд, цене, кратком описании и составе.
2. База клиентов. Содержится информация о фамилии, имени, его контактный телефон, адрес. Имеется информация потраченных за всё время средств, при достижении определённой суммы клиенту предоставляется скидка 10% на дальнейшие заказы.

Вариант 3. Тема: Театр

Предстоит организационно-финансовая работа, связанная с привлечением актёров

Реализовать структуры:

1. База спектаклей. В базе содержится информация о названии, жанре и бюджете спектакля.
2. База актёров. Содержится информация о фамилии, имени, стаже работы, контактном телефоне и стоимости найма актёра, выручка актёра. При найме актёра на работу с конкретным спектаклем фиксируется дата заключения договора и из бюджета спектакля вычитается заработная плата актёра.

Вариант 4. Тема: Оптовая продажа товаров

Предстоит работа с оптовой продажей различных товаров.

Реализовать структуры:

1. База товаров. В базе содержится информация о наименовании товара, оптовой цене, справочной информации, количестве товара на складе.
2. База покупателей. Содержится информация о фамилии, имени, отчестве и контактном телефоне, номере договора, наименовании товара, количестве заказанного товара, городе проживания. Если количество товара превышает имеющийся запас на складе, заказ отменяется.

Вариант 5. Тема: Автоматизация склада

Предстоит работа с автоматизацией управления складом.

Реализовать структуры:

1. База товаров. В базе содержится информация о наименовании товара, количестве товара на складе, дате поставки, дате последней продажи.
2. База заказов. Содержится информация о имени, фамилии заказчика, его контактном телефоне, дате и времени заказа, списке заказанных товаров и количестве каждого товара, адресе доставки. Если количество товара превышает имеющийся запас на складе, заказ отменяется.

Вариант 6. Тема: Кинотеатр

Необходимо создать систему учета фильмов в кинотеатре.

Реализовать структуры:

1. База фильмов. Содержит информацию о названии фильма, жанре, годе выхода, режиссере и актерах. Для каждого фильма должна быть информация о его доступности в данный момент (например, он может быть в прокате или уже закончил показ).
2. База зрителей. Содержит информацию об имени, фамилии, контактном телефоне и возрасте зрителя. Для каждого зрителя должна быть информация о билетах, которые он зарезервировал для просмотра фильмов.

Вариант 7. Тема: Интернет-магазин

Предстоит работа с интернет-магазином, который продает продукты питания.

Реализовать структуры:

1. База продуктов. В базе содержится информация о наименовании продукта, его цене, производителе и количестве, имеющемся в наличии. Для каждого продукта должна быть информация об его категории (например, фрукты, овощи, молочные продукты).
2. База покупателей. Содержит информацию об имени, фамилии, адресе и контактном телефоне покупателя. Для каждого покупателя должна быть информация о заказанных им продуктах и количестве каждого продукта. Если покупатель закажет товар, которого больше нет в наличии, выведется ошибка.

Вариант 8. Тема: Социальная сеть

Предстоит работа с социальной сетью, где пользователи могут добавлять друзей, создавать сообщества и обмениваться сообщениями.

Реализовать структуры:

1. База сообществ. Содержит информацию о наименовании сообщества, тематике и создателе сообщества. Сообщество содержит информацию о количестве вступивших пользователей.
2. База пользователей. В базе содержится информация об имени, фамилии, дате рождения, адресе и контактном телефоне пользователя. Для каждого пользователя должна быть информация об сообществах, в которых он состоит.

Вариант 9. Тема: Транспортировка грузов

Разработать программу для учета транспортировки грузов.

Реализовать структуры:

1. База транспортных средств. В базе содержится информация о транспортных средствах, доступных для перевозки грузов, включая их тип (например, грузовик, поезд, корабль), грузоподъемность и местоположение.
2. База грузов. Содержит информацию о грузах, которые необходимо перевезти, включая их вес, тип и место отправления (например, город Минск, Гродно, Москва), имя, фамилия и контактный телефон получателя. При совпадении грузоподъемности транспортного средства и веса груза, транспортное средство привязывается к грузу.

Вариант 10. Тема: Услуги ЖКХ

Необходимо создать программу для учета автомобилей в автосервисе.

Реализовать структуры:

1. База заявок. В базе содержится информация о адресе заявки, описании проблемы, категории проблемы (например, санитарное состояние территории, электроснабжение, общестроительные работы), статус заявки.
2. База клиентов. Содержит информацию об имени, фамилии, городе проживания и контактном телефоне. Клиенты оставляют заявки, указывая все нужные требования. В зависимости от категории проблемы к заявке приписывается срок выполнения.

Вариант 11. Тема: Школа

Разработать программу учета учеников и учителей в школе.

Реализовать структуры:

1. База учителей. Содержит информацию об имени, фамилии, предметах, которые он преподает, и контактной информации учителя.

2. База учеников. В базе содержится информация об имени, фамилии, дате рождения и контактном телефоне ученика. Для каждого ученика должна быть информация об учителях и предметах, которые он изучает.

Вариант 12. Тема: Ресторан

Предстоит работа с организацией брони столов в ресторане.

Реализовать структуры:

1. База столов. В базе содержится информация о размере стола (маленький, средний, большой), номере столика (например, №15), стоимости и доступности в данный момент (при бронировании стола, он становится недоступным).
2. База посетителей. Содержится информация об имени, фамилии, отчестве посетителя и его контактный телефон. Для каждого клиента должна иметься информация о забронированных им столиках со временем бронирования.

Вариант 13. Тема: Стиральная комната

Предстоит работа с организацией доступности стиральных машин в стиральной комнате.

Реализовать структуры:

1. База стиральных машин. В базе содержится информация о типе машины (обычная, большая), номере машины (например, №4) и стоимости.
2. База клиентов. Содержится информация об имени, фамилии, отчестве клиента и его контактный телефон. Для каждого клиента должна иметься информация о забронированных им машинах со временем бронирования.

Вариант 14. Тема: Магазин

Предстоит работа с организацией покупок в магазине.

Реализовать структуры:

1. База товаров. В базе содержится информация о наименовании товара, его цене и количестве.
2. База покупателей. Содержится информация об имени, фамилии, отчестве клиента, бюджете и его контактный телефон. Для каждого клиента должна иметься информация о купленных им товарах с датой их покупки. Если общая стоимость заказа превышает бюджет клиента, выводится соответствующее сообщение.

Вариант 15. Тема: Туристическое агентство

Реализовать структуры:

1. База туров. В базе должно быть содержаться название тура, страна, длительность тура, стоимость и доступность в данный момент.
2. База клиентов. Содержит информацию о имени, фамилии, контактном телефоне, электронной почте и дате рождения клиента. Для каждого клиента должна быть информация о купленных им турах с указанием даты вылета. Учитывая длительность тура рассчитывается дата прибытия.

Вариант 16. Тема: Книжный магазин

Реализовать структуры:

1. База книг. В базе содержится информация об авторе, названии, годе выпуска, жанре и стоимости книги.
2. База клиентов. Содержит информацию о имени, фамилии, контактном телефоне и электронной почте клиента. Для каждого клиента должна быть информация о купленных им книгах, включая дату покупки и номер заказа. Дополнительно содержится информация о подписке на авторов, книги этих авторов автоматически добавляются в заказ.

Задание 2.

Реализовать следующий функционал для работы с структурами:

1. Автоматическое заполнение первой структуры
2. Регистрация (заполнение пользователем второй структуры)
3. Вывод структур на экран.

Задание 3.

Написать программу, в которой нужно создать объединение для хранения двух типов данных (например, целых чисел и логических значений) и написать функцию для вывода этих значений на экран.

Задание 4.

Написать программу, в которой нужно использовать объединение для хранения значения переменной в разных форматах (например, как целое число или как массив байтов).

Задание 5.

Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить название планеты Солнечной системы, с использованием перечислений.

Контрольные вопросы

1. Определение структуры.
2. Способы получения доступа к элементам структуры.
3. Определение массива структуры.
4. Что такое UNION в C++? Как он отличается от структуры?
5. Как объявить UNION в C++?
6. Какие операции можно выполнять с UNION в C++?
7. Какие могут быть ограничения при использовании UNION для создания объединений?
8. Что такое ENUM в C++? Как он используется для определения констант?
9. Как объявить ENUM в C++?
10. Какие операции можно выполнять с ENUM в C++?

Дополнительные задания

Дополнительное задание 1.

Добавить следующую структуру:

Вариант 1. База сотрудников гостиницы. Содержит информацию о фамилии, имени, отчестве сотрудника, его телефоне для связи, должности, окладе и дате приёма на работу. При брони клиентами номеров сохраняется информация, какой сотрудник оформлял бронь.

Вариант 2. База отзывов. После заказа еды клиент может оставить отзыв на конкретное блюдо. Отзыв содержит в себе текстовую часть и оценку по пятибалльной шкале. Если рейтинг блюда составляет менее двух, при дальнейших заказах клиентам приходит предупреждение о низком рейтинге выбранного ими блюда.

Вариант 3. База зрителей. При покупке билета, зритель вносится в базу. Содержится информация о выбранном спектакле, месте в зале. Клиент не может выбрать место в зале, если оно уже куплено другим. При достижении определённой отметки заказанных билетов на спектакль, актёрам, участвующим в спектакле, выписывается премия.

Вариант 4. База пунктов выдачи. Содержится информация о местонахождении, контактном номере, времени работы. При каждом заказе идёт выбор, в какой пункт выдачи его направлять. Выбирается тот пункт, в котором проживает сам клиент. Если пункта выдачи в его городе нет, клиенту приходит соответствующее сообщение и даётся самостоятельное право выбора пункта.

Вариант 5. База поставщиков. Содержится информация о наименовании поставщика, контактных данных, списке товаров, которые он предлагает. В случае, когда товар на складе начинает заканчиваться, приходит запрос поставщикам, которые поставляют заканчивающийся товар. Фиксируется дата поставки и количество поставляемого товара.

Вариант 6. База сеансов. Содержит информацию о времени начала показа каждого фильма, номере зала и количестве свободных мест на данный момент. Для каждого сеанса должна быть информация о фильме, который будет показан, и доступности билетов на данный сеанс.

Вариант 7. База курьеров. Содержит информацию о фамилии, имени и отчестве курьера, стаже работы. При оформлении заказа пользователем курьеру присваивается задача доставки. Клиенты могут оценивать работу курьера по пятибалльной шкале. Средняя оценка содержится в информации о курьере.

Вариант 8. База постов. В этой структуре содержится информация о посте, написанном пользователем в сообществе. Она содержит информацию о отправителе сообщения, дате отправки, тексте поста. Пост прикрепляется к определенному сообществу и может быть оставлен только тем пользователем, который в этом сообществе состоит.

Вариант 9. База водителей. Содержит информацию о фамилии, имени и отчестве водителя, стаже работы и категории прав. Водитель может управлять только тем транспортным средством, на которые у него имеются права. В базе находится информация о местоположении водителя, если оно совпадает с местоположением груза, водитель берётся за перевозку.

Вариант 10. База работников. Содержит информацию о фамилии, имени и отчестве работника, специализации (например, инженер). Работникам приходят заявки, соответствующие их специализации. Работник не может брать на себя больше пяти заявок одновременно, если к нему поступает шестая, она автоматически переходит в статус “в ожидании”.

Вариант 11. База учебного плана. Структура должна содержать информацию о классе (например, 11 “А”) и списке предметов, которые будут изучаться в данном году. Каждый предмет содержит информацию об учителе, который будет преподавать этот предмет, и списке учеников, которые будут изучать этот предмет.

Вариант 12. База меню. Содержит информацию о блюдах, напитках и их стоимости в ресторане. Кроме того, для каждого блюда и напитка должна быть информация о его доступности в данный момент (например, если продукты закончились, блюдо временно может быть недоступно). Выбор меню для посетителя становится доступен после брони стола. Выбранные позиции из меню сохраняются в базе клиента.

Вариант 13. База расписания. Содержит информацию о расписании использования стиральных машин в стиральной комнате. Для каждой машины должно быть указано время, когда она будет доступна для использования, а также время, когда она будет занята. Кроме того, для каждой машины должна быть информация о ее доступности в данный момент (например, если машина вышла из строя, она может быть временно недоступна). При попытке брони клиентом машины происходит проверка по расписанию.

Вариант 14. База работников. Содержит информацию о фамилии, имени и отчестве сотрудника, его телефоне для связи, должности, окладе и дате приёма на работу. Кроме того, нужно хранить информацию о том, какой уровень сертификации у него есть. При покупке товара сохраняется информация о работнике, оформляющем покупку.

Вариант 15. База сотрудников. В базе должна содержаться информация о сотрудниках туристического агентства, включая их имена, фамилии, должности, контактные данные и даты начала работы. Для каждого сотрудника должна быть информация о турах, которые он продал.

Вариант 16. База отзывов. В базе должны содержаться отзывы клиентов о книгах, которые они купили в магазине. Для каждого отзыва должна быть информация о клиенте, который его написал, а также о книге, на которую отзыв написан. Отзыв должен содержать оценку книги по шкале от 1 до 5, а также текстовый комментарий.

Дополнительное задание 2.

Напишите программу, которая позволяет пользователю выбрать тип фигуры (круг, квадрат или треугольник) и вводить параметры этой фигуры (радиус, стороны и т.д.) с помощью объединения (union) в C++. Затем программа должна вычислить площадь выбранной фигуры и вывести ее на экран.

Дополнительное задание 3.

Допустим, у нас есть программа, которая управляет автомобилем. Водитель может выбрать режим езды, который определяет, как автомобиль будет реагировать на действия водителя. Например, в режиме спорт автомобиль будет более резко реагировать на управление, а в режиме экономия - экономнее расходовать топливо. Использовать enum.

Лабораторная работа № 2. Создание меню программы. Разбиение программы на файлы. STL контейнеры. Array

Цель: ознакомиться с процессом создания меню программы, реализовать доступ ко всему необходимо функционалу, изучить принципы разбиение программы на файлы. Узнать что представляют из себя STL контейнеры, изучить средства работы с контейнером <array>.

Теоретические сведения № 1

Создание меню в базах данных необходимо для обеспечения удобного и интуитивно понятного интерфейса для пользователя, позволяет пользователю выбирать необходимые ему функции и операции из списка доступных вариантов. Это может существенно упростить работу с базой данных и снизить вероятность ошибок при вводе команд.

Меню может быть использовано для ограничения доступа к определенным функциям базы данных только для определенных пользователей. Например, администратор базы данных может иметь доступ ко всем функциям, в то время как обычный пользователь может иметь доступ только к просмотру и редактированию своих записей.

Управление целостностью данных. Создание меню может помочь в обеспечении целостности данных, поскольку пользователи могут выбирать только определенные операции и функции, которые были предварительно проверены и одобрены разработчиками базы данных. Это может предотвратить случайное удаление или изменение данных.

Управление производительностью базы данных. Меню может быть использовано для оптимизации производительности базы данных, поскольку пользователи могут выбирать только те функции, которые им действительно нужны, вместо того, чтобы выполнять все операции вручную. Это может существенно снизить нагрузку на базу данных и ускорить ее работу.

В целом, создание меню в базах данных C++ может помочь в обеспечении удобного и безопасного доступа к данным, а также улучшить производительность и целостность базы данных.

```
#include <iostream>
#include <windows.h>
using namespace std;

int choice;

void menu() {
    cout << "1. Добавить пользователя" << endl;
    cout << "2. Удалить пользователя" << endl;
    cout << "3. Просмотр списка пользователей" << endl;
    cout << "4. Поиск пользователей" << endl;
    cout << "5. Выход" << endl;
```

```

    cout << "Введите ваш выбор: ";
    cin >> choice;
}

int main() {
setlocale(LC_CTYPE, "rus");
do {
    menu();
    switch (choice) {
        case 1:
            system("cls"); // очистка консоли // ...
            system("pause"); // задержка консоли
            system("cls");
            break;
        case 2:
            system("cls");
            // ...
            system("pause");
            system("cls");
            break;
        case 3:
            system("cls");
            // ...
            system("pause");
            system("cls");
            break;
        case 4:
            system("cls");
            // ...
            system("pause");
            system("cls");
            break;
        case 5:
            system("cls");
            cout << "Выход..." << endl;
            system("pause");
            system("cls");
            break;
        default:
            system("cls");
            cout << "Неправильный выбор" << endl;
            system("pause");
            system("cls");

```

```
        break;
    }
} while (choice != 5);

return 0;
}
```

В C++ программы часто состоят из нескольких файлов. Это может быть полезно для улучшения организации кода и уменьшения сложности программы. Разбиение программы на несколько файлов может помочь сделать код более читаемым, уменьшить время компиляции и повторное использование кода.

Существует несколько способов разбиения программы на файлы в C++. Один из самых распространенных способов - это использование заголовочных файлов и файлов реализации.

Заголовочный файл содержит объявления функций, классов и переменных, которые используются в программе. Обычно он имеет расширение .h или .hpp. Заголовочные файлы обычно не содержат реализации функций, только их объявления.

Файл реализации содержит реализацию функций и классов, объявленных в заголовочном файле. Обычно он имеет расширение .cpp. Файл реализации должен включать заголовочный файл, чтобы компилятор знал, какие функции и классы нужно реализовать.

Разбиение программы на несколько файлов может помочь сделать код более читаемым и уменьшить сложность программы. Однако необходимо следить за тем, чтобы файлы были логически связаны друг с другом и правильно организованы в структуру проекта.

Теоретические сведения № 2

Синтаксис STL основан на использовании таких синтаксических конструкций языка C++ как шаблоны (templates) классов и шаблоны функций.

По своей сути STL — это сложный набор шаблонных классов и функций, реализующих многие популярные и часто используемые структуры данных и алгоритмы.

Центральным понятием STL, вокруг которого крутится всё остальное, это контейнер. **Контейнер** — это набор некоторого количества обязательно однотипных элементов, упакованных в контейнер определенным образом. Простейшим прототипом контейнера в классическом языке C++ является массив. Тот способ, которым элементы упаковываются в контейнер и определяет тип контейнера и особенности работы с элементами в таком контейнере. STL вводит целый ряд разнообразных типов контейнеров, основные из них:

- последовательные контейнеры — вектор (vector), двусвязный список (list), дэк (deque);
- ассоциативные контейнеры — множества (set), хэш-таблицы (map);

Контейнер	Описание	Требуемый заголовок
array	Массив	<array>
vector	Динамический массив	<vector>
deque	Двунаправленная очередь	<deque>
forward_list	Односвязный список	<forward_list>
list	Двусвязный список	<list>
set	Множество уникальных элементов	<set>
map	Хранит пары “ключ\значение”, где каждый ключ может быть ассоциирован с двумя и более значениями	<map>

Массив — структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона.

std::array — это фиксированный массив, который не распадается в указатель при передаче в функцию. std::array определяется в заголовочном файле array, внутри пространства имен std. В отличие от стандартных фиксированных массивов, в std::array вы не можете пропустить (не указывать) длину массива.

Объявим переменную std::array :

```
#include <array>

int main()
{
    std::array<int,6> numbers; // объявляем массив значений int длиной 6
}
```

Чтобы инициализировать контейнер определенными значениями, можно использовать инициализатор - в фигурных скобках передать значения элементам контейнера:

```
#include <array>

int main()
{
    std::array<int, 6> numbers{ 2, 3, 4, 5, 6, 7}; // состоит из 6 чисел int
}
```

Для доступа к элементам контейнера array можно применять тот же синтаксис, что при работе с массивами - в квадратных скобках указывать индекс элемента, к которому идет обращение:

```
#include <array>
#include <iostream>

using namespace std;

int main()
{
    array<int, 6> numbers{ 2, 3, 4, 5, 6, 7 };
    int n = numbers[3];
    cout << "n = " << n << std::endl; // получаем значение элемента n = 5
    numbers[3] = 12; // меняем значение элемента
    cout << "numbers[3] = " << numbers[3] << endl; // numbers[3] = 12
}
```

Основные функции array

В контейнер array нельзя добавлять новые элементы, так же как и удалять уже имеющиеся. Основные функции типа array, которые мы можем использовать:

- **size()**: возвращает размер контейнера
- **at(index)**: возвращает элемент по индексу index
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент
- **fill(n)**: присваивает всем элементам контейнера значение n

```
#include <iostream>
#include <array>

using namespace std;

int main()
```

```

{
    array<int, 3> numbers{ 1, 2, 3 };
    int second = numbers.at(1);
    int first = numbers.front();
    int last = numbers.back();
    cout << "Length: " << numbers.size() << endl; //Length: 3
    cout << "Second element: " << second << endl; //Second element: 2
    cout << "First element: " << first << endl; //First element: 1
    cout << "Last element: " << last << endl; //Last element: 3
    numbers.fill(0);
    cout << "All elements: "; //All elements: 0 0 0
    for (int i{0}; i < numbers.size(); i++)
    {
        cout << numbers[i] << " ";
    }
}

```

Задания к лабораторной работе № 2

Задание 1.

Реализовать полноценное меню, показывающее весь функционал, описанный в предыдущей лабораторной работе.

```

1. Регистрация нового клиента
2. Вход в клиентскую систему
3. Просмотр каталога
4. Вход в меню сотрудников
0. Выход
Введите ваш выбор: _

```

Задание 2.

Разбить программу на файлы. Предлагаем разбить программу на следующие файлы:

1. main.cpp - содержит функцию main, в которой реализован switch case с меню программы
2. func.cpp - содержит определение функций
3. func.h - содержит объявление функций

Задание 3.

Написать программу, в которой нужно найти сумму минимального и максимального элементов массива, с выводом на экран.

Задание 4.

Написать программу, в которой нужно создать два массива , объединить их и вывести на экран сумму элементов.

Контрольные вопросы

1. Для чего необходимо создание меню в базах данных?
2. Принципы разбиения программы на файлы.
3. Что такое контейнер?
4. Типы контейнеров.
5. Что такое array в C++?
6. Как объявить array в C++?
7. Какие операции можно выполнять с array в C++?
8. Как array может быть использован для хранения и обработки данных в C++?

Дополнительные задания

Дополнительное задание 1.

Учесть при создании меню функционал третьей структуры, созданной в дополнительном задании к лабораторной работе №1.

Дополнительное задание 2.

Улучшить визуальное представление меню, используя изменение цвета консоли. (Приложение 1).

Дополнительное задание 3.

Напишите программу, которая запрашивает у пользователя 10 целых чисел и сохраняет их в контейнере array в STL C++. Затем программа должна вычислить среднее арифметическое этих чисел и вывести на экран все числа, которые больше среднего арифметического.

Лабораторная работа № 3. Строки типа string. Фильтрация данных. Vector

Цель: ознакомиться с основными методами работы со строками типа string, ознакомиться с алгоритмом поиска подстроки в строке, научиться работать с контейнером <vector>.

Теоретические сведения № 1

Для хранения строк в C++ часто используют тип std::string из модуля <string>. Строки типа string имеют ряд преимуществ:

- возможность обработки строк стандартными операторами C++ (=, +, ==, < и т.п.);
- обеспечение лучшей надежности (безопасности) программного кода;
- возможность использования методов класса string для работы со строками;
- автоматическое управление памятью;
- возможность использования строки, как самостоятельного типа данных.

Для использования строк типа string в C++ необходимо подключить заголовочный файл string.

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

int main() {
    std::string str = "Hello, world!";
    std::cout << str << std::endl;
    return 0;
}
```

Ввод и вывод строк типа string можно осуществлять стандартными потоками ввода-вывода cin и cout.

```
string str;
cin >> str; //hello world
cout << str; //hello
```

Следует обратить внимание на то, что строка читается до первого пробела.

Альтернативный способ ввода строки - использование функции **getline()**. Она имеет следующий прототип:

```
istream& getline(istream& is, string& str, char delim);
```

где:

- is - входной поток, из которого необходимо считать строку;
- str - объект типа `string`, в который нужно записать считанную строку;
- delim - опциональный параметр, который указывает на символ-разделитель, до которого нужно считывать строку. По умолчанию разделитель ставится на символ переноса строки.

В данном примере, getline считывает всю строку до символа новой строки из потока ввода cin. Введенная строка сохраняется в переменной str.

```
string str;  
getline(cin, str); //hello world  
cout << str; //hello World
```

Функция getline() является удобным способом ввода строк в C++ типа string, но при ее использовании есть некоторые нюансы.

Особенности и проблемы:

- Функция getline() может считать пустую строку;
- Если считываемый текст содержит символ-разделитель, то getline() остановится на этом символе, но не включит его в строку;
- Значение разделителя может быть любым символом, в том числе и нестандартным, но при этом нужно быть осторожным, чтобы символ-разделитель не совпадал с вводимыми данными;
- Если перед вызовом getline() в потоке остался символ переноса строки, он будет являться пустой строкой для этого вызова, что может привести к ошибкам.

Чтобы это исправить, можно использовать функцию **cin.ignore()**. Функция cin.ignore используется для очистки символов во входном потоке для следующего считывания. Она имеет следующий прототип:

```
istream& ignore(streamsize n = 1, int delim = EOF);
```

где:

- n – количество символов, которое нужно пропустить;
- delim – опциональный параметр, который указывает, до какого символа нужно пропустить символы. По умолчанию используется значение EOF.

Метод cin.ignore() предназначен для извлечения символов из входного потока (cin). Извлеченные символы игнорируются и не используются. Метод cin.ignore() может принимать один или два аргумента. Если метод вызывается с одним аргументом, то он игнорирует один символ. Если метод вызывается с двумя аргументами, то он игнорирует до n символов или до тех пор, пока не встретит символ разделитель (второй параметр).

cin.ignore(); игнорирует один символ.
cin.ignore(10); игнорирует до 10 символов.
cin.ignore(10, ' '); игнорирует до 10 символов или до тех пор, пока не будет достигнут пробел.

cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); игнорирует все символы до символа новой строки '\n' или до тех пор, пока не будет достигнут предел числового типа std::streamsize.

Пример использования:

```
string str;
cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка входного потока
getline(cin, str); //input till '\n'

//hello world
cout << str; //hello world
```

Это позволит проигнорировать оставшиеся символы в буфере ввода, включая символ новой строки, которые могут вызвать ошибку в использовании getline().

Существует ряд удобных методов класса string для работы со строками:

- append() -объединение строк;

```
std::string str = "Hello";
str.append(", world!");
cout << str; //Hello, world!
```

- assign() - присваивание строки;

```
std::string str1 = "Hello";
std::string str2 = "world";
str1.assign(str2);
cout << str1; // str1 = "world"
```

- insert() - вставка строки с заданной позиции;

```
std::string str = "Hello";
str.insert(5, ", world!");
cout << str; // "Hello, world!"
```

- replace() - вызываемая строка заменяется другой строкой. Есть возможность задать позицию и количество символов в вызываемой строке, которые нужно заменить другой строкой;

```
std::string str = "Hello, world!";  
str.replace(5, 6, "C++");  
cout << str; // HelloC++d!
```

- `erase()` - удаление указанного количества символов строки начиная с указанной позиции;

```
std::string str = "Hello, world!";  
str.erase(5, 7);  
cout << str; // Hello!
```

- `find()` - поиск подстроки в строке, возвращает индекс первого вхождения подстроки в строку. Если подстрока не найдена, то метод возвращает -1;

```
std::string str = "Hello, world!";  
int pos = str.find("world");  
cout << pos; // 7
```

- **`string::npos`** -это статическая константа в классе `std::string`, которая используется для обозначения конца строки. Она имеет значение -1, что означает, что она может быть использована для проверки наличия символа в строке. Если метод `find()` не находит символ в строке, он возвращает значение `string::npos`

```
std::string str = "Hello, world!";  
int pos = str.find("c++");  
cout << pos; // -1
```

- `size()` - получение размера строки;

```
std::string str = "Hello, world!";  
int size = str.size();  
cout << size; // 13
```

- `empty()` - проверка на пустоту.

```
std::string str = "";  
bool is_empty = str.empty();  
cout << is_empty; // 1
```


Фильтрация данных - это процесс выборки данных из набора данных на основе определенных критериев. Фильтрацию можно производить по наличию определенной подстроки в строке.

Задача поиска подстроки в строке реализуется очень просто, если использовать встроенные функции для работы со строками типа string. Например, вот пример кода для поиска позиции вхождения подстроки "pa" в строке:

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

using namespace std;
int main()
{
    string s = "parapapa";
    int i = 0;
    for (i = s.find("pa", i++); i != string::npos; i = s.find("pa", i + 1))
        cout << i << " "; //0 4 6
}
```

Реализация этой же задачи, но при помощи массива типа char будет гораздо длиннее, но алгоритм все же достаточно прост:

Есть две строки. Например "Hello world" и "lo". Работать будем в два цикла:

1. Первый будет выполнять проход по всей строке, и искать местоположение **первой буквы** искомой строки ("lo").

2. Второй, начиная с найденной позиции первой буквы – сверять, какие буквы стоят после неё и сколько из них подряд совпадают.

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							

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

using namespace std;
```

```

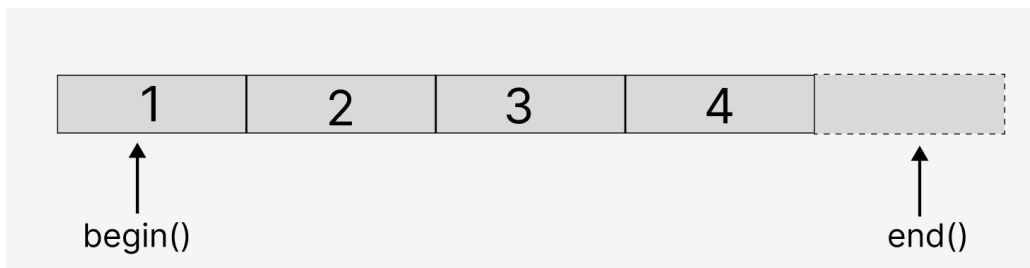
int main() {
setlocale(LC_CTYPE, "rus");
char str[] = "abababbbabbbb"; // строка
char substr[] = "ab"; // подстрока, которую мы ищем
// функция strlen из библиотеки cstring получает длину строки.
int len_str = strlen(str); // длина строки
int len_substr = strlen(substr); // длина подстроки
int count = 0; // счетчик количества вхождений
for (int i = 0; i <= len_str - len_substr; i++) { // пока есть возможность поиска
bool found = true;
for (int j = 0; j < len_substr; j++) { // цикл проверяет, соответствует ли текущая
подстрока искомой подстроке
if (str[i + j] != substr[j]) {
found = false;
break;
}
}
if (found) {
count++;
cout << i << " ";
}
}
if (count == 0) {
cout << "Вхождений не найдено" << endl;
}
else {
cout << "\nОбщее количество вхождений: " << count << endl;
}
// вывод:
//0 2 4 8
//Общее количество вхождений: 4
return 0;
}

```

Теоретические сведения № 2

Векторы аналогичны динамическим массивам с возможностью автоматического изменения размера при вставке или удалении элемента, а их хранение автоматически обрабатывается контейнером. Векторные элементы размещаются в непрерывной памяти, чтобы к ним можно было обращаться и перемещаться с помощью итераторов. В векторах данные вставляются в конце.

В стандартной библиотеке C++ *вектором* называется динамический массив, обеспечивающий быстрое добавление новых элементов в конец и меняющий свой размер при необходимости.



Сначала для создания вектора нам понадобится подключить библиотеку : `<vector>`.

Чтобы объявить вектор, нужно пользоваться конструкцией ниже:

```
vector < тип данных > <имя вектора>;
```

Элементы вектора должны быть одинакового типа, и этот тип должен быть известен при компиляции программы. Он задаётся в угловых скобках после `std::vector`: например, `std::vector<int>` — это вектор целых чисел типа `int`, а `std::vector<std::string>` — вектор строк.

Мы можем инициализировать вектор одним из следующих способов:

```
std::vector<int> v1;           // пустой вектор
std::vector<int> v2(v1);      // вектор v2 - копия вектора v1
std::vector<int> v3 = v1;     // вектор v3 - копия вектора v1
std::vector<int> v4(5);       // вектор v4 состоит из 5 чисел, каждое число равно 0
std::vector<int> v5(5, 2);    // вектор v5 состоит из 5 чисел, каждое число равно 2
std::vector<int> v6{ 1, 2, 4, 5 }; // вектор v6 состоит из чисел 1, 2, 4, 5
std::vector<int> v7 = { 1, 2, 3, 5 }; // вектор v7 состоит из чисел 1, 2, 4, 5
```

Для обращения к элементам вектора можно использовать разные способы:

- **[index]**: получение элемента по индексу (также как и в массивах), индексация начинается с нуля
- **at(index)**: функция возвращает элемент по индексу
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент

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

using namespace std;

int main()
{
    vector<int> numbers{ 1, 2, 3, 4, 5 };
}
```

```

int first = numbers.front();
int last = numbers.back();
int second = numbers[1];
int third = numbers.at(3);
cout << "first: " << first << endl; //first: 1
cout << "second: " << second << endl; //second: 2
cout << "last: " << last << endl; // last: 5
cout << "third: " << third << endl; //third: 4

    numbers[0] = 6; // изменяем значение
for (int n : numbers)
    cout << n << "t"; // 6 2 3 4 5
cout << endl;
return 0;
}

```

Чтобы узнать общее количество элементов в векторе, можно воспользоваться функцией **size**:

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> numbers{ 1, 2, 3, 4, 5 };
    cout << numbers.size() << "n"; // 5
    return 0;
}

```

С помощью функции **push_back()** мы можем добавить ячейку в конец вектора. А функция **pop_back()** все делает наоборот — удаляет одну ячейку в конце вектора.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> numbers{ 1, 2, 3, 4, 5 };
    numbers.push_back(6);
    numbers.push_back(7);
    numbers.push_back(8);
}

```

```

for (int i = 0; i < numbers.size(); i++) {
    cout << numbers[i] << " ";
} // 1 2 3 4 5 6 7 8
cout << endl;
numbers.pop_back();
for (int i = 0; i < numbers.size(); i++) {
    cout << numbers[i] << " ";
} // 1 2 3 4 5 6 7
cout << endl;
return 0;
}

```

Добавление элементов в другие части вектора или их удаление неэффективно, так как требует сдвига соседних элементов. Поэтому отдельных функций, например, для добавления или удаления элементов из начала у вектора нет.

Удалить все элементы из вектора можно с помощью функции **clear**.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> numbers{ 1, 2, 3, 4, 5 };
    cout << "Before clear:" << endl; // 1 2 3 4 5
    for (int i = 0; i < numbers.size(); i++) {
        cout << numbers[i] << " ";
    }
    numbers.clear();
    cout << endl;
    cout << "After clear:" << endl; // вывод пустой строки
    for (int i = 0; i < numbers.size(); i++) {
        cout << numbers[i] << " ";
    }
    return 0;
}

```

Задания к лабораторной работе № 3

Задание 1.

В вашей бд сделайте так, чтобы строковые поля в структуре могли состоять из нескольких строк. Перепишите ввод этих элементов используя `getline`.

Например: ФИО клиента записывается не как три отдельные поля в структуре: Фамилия, Имя, Отчество, а как одно и вводится с клавиатуры сразу.

```
Введите имя
Julia
Введите фамилию
Bobr
Введите отчество
Stepanovna
Введите мобильный телефон
8029111111
```

```
Введите ФИО
Julia Bobr Stepanovna
Введите мобильный телефон
8029111111
```

Задание 2.

Для вашей бд осуществите фильтрацию по строчным элементам структуры (например поле имя клиента или название продукта и тд). Фильтрация должна работать следующим образом: при вводе с клавиатуры подстроки, в консоль выводятся элементы массива структур, содержащие данную подстроку в поле, по которому происходит фильтрация.

Пример: Вариант 1. Гостиница.

Фильтрация комнат по уровню комфорта.

```
1. Вывести каталог
2. Отфильтровать каталог по уровню комфорта
3. Выход в меню
Введите ваш выбор: 2
```

```
Доступные уровни комфорта:
standar
junoirsuite
luxe

Введите уровень комфорта: luxe
```

```
Номер № 252
Уровень комфорта: luxe
Цена: 1300
Доступен

Номер № 220
Уровень комфорта: luxe
Цена: 1700
Доступен

Номер № 324
Уровень комфорта: luxe
Цена: 1500
Доступен
```

Задание 3.

Написать программу, в которой нужно создать два вектора и объединить их, вывести на экран все уникальные элементы.

Задание 4.

Создать матрицу из массивов vector. В данной матрице удалить любую одну строку с наибольшим количеством нулей.

Контрольные вопросы

1. Назовите основные методы работы со строками string.
1. Преимущества строк типа string.
2. Особенности и проблемы в использовании getline().
3. Что из себя представляет метод cin.ignore()?
4. Что такое фильтрация данных?
5. Алгоритм поиска подстроки в строке.
6. Что такое vector в C++?
7. Как объявить и инициализировать vector в C++?
8. Как добавить элемент в конец vector в C++?
9. Как удалить элемент из vector в C++?
10. Как получить доступ к элементу vector по индексу в C++?
11. Как получить размер vector в C++?

Дополнительные задания

Дополнительное задание 1.

В вашей БД реализовать функцию проверки номера телефона на валидность при вводе. Используя методы для работы со строками string проверьте, что введенный номер соответствует одному из способов представления:

+375(XX)XXX-XX-XX или 80(XX)XXX-XX-XX

Дополнительное задание 2.

Напишите программу, которая запрашивает у пользователя последовательность целых чисел и сохраняет их в контейнере `vector`. Затем программа должна удалить все дубликаты из этой последовательности и вывести на экран новую последовательность.

Дополнительное задание 3.

Осуществить реализацию стека или очереди при помощи вектора

Лабораторная работа № 4 Двоичные и текстовые файлы. Запись/чтение массива структур в файл. Deque

Цель: ознакомиться с отличиями между двоичными и текстовыми файлами; изучить основные методы работы с файлами, научиться работать с контейнером <deque>

Теоретические сведения № 1

Файл - это набор данных, который хранится на компьютере или другом устройстве хранения данных. Файлы используются для хранения информации, которую можно обрабатывать или использовать позже. Они обычно имеют уникальное имя и расширение, которое указывает на тип данных, содержащихся в файле.

Текстовые файлы - это файлы, которые содержат символы, которые можно прочитать и интерпретировать как текст. Такие файлы, как правило, содержат данные, которые могут быть прочитаны и изменены с помощью текстовых редакторов, таких как блокнот.

Двоичные файлы - это файлы, которые содержат данные в бинарном формате, то есть данные, которые не могут быть прочитаны и поняты как текст. Такие файлы, как правило, содержат данные, которые могут быть прочитаны и обработаны только компьютером, например, изображения, звуковые файлы, исполняемые файлы и другие файлы, содержащие бинарные данные.

Двоичные файлы и текстовые файлы отличаются не только типом данных, но и методами работы с ними. Например, если мы записываем цифру «4» в текстовый файл, то она записывается как символ, и для ее хранения нужен один байт. Текстовый файл, содержащий запись: «145687», будет иметь размер шести байт.

Однако, если записать целое число 145 687 в двоичный файл, то он будет иметь размер четыре байта, так как именно столько необходимо для хранения данных типа `int`. Таким образом, двоичные файлы позволяют более эффективно использовать память компьютера, поскольку они хранят данные в бинарном формате, который занимает меньше места, чем текстовый формат.

Работа с файлами

Для работы с файлами в программировании используются специальные типы данных, называемые потоками. Поток **ifstream** служит для работы с файлами в режиме чтения, а поток **ofstream** - в режиме записи. Если нужно работать с файлом как с потоком ввода-вывода, то используется поток **fstream**.

Класс ifstream

Для чтения файлов в C++ используются потоки данных (streams). В данном случае, для работы с текстовыми файлами, используются объекты класса `ifstream`.

Примеры открытия файла для чтения:

1. Открытие файла с помощью метода `open()`:

```

#include <fstream>

int main() {
std::ifstream file;
file.open("C:/file.txt"); // Открытие файла с помощью метода open()
if (file.is_open()) { // Проверка открытия файла
    // Файл успешно открыт, можно читать из него данные
}
else {
    // Произошла ошибка при открытии файла
    return -1;
}

file.close(); // Закрытие файла
return 0;
}

```

2. Открытие файла с помощью конструктора:

```

#include <fstream>
using namespace std;

int main()
{
    ifstream file("C:/file.txt"); // Открытие файла с помощью конструктора
    if (!file) { // Проверка открытия файла
        // Произошла ошибка при открытии файла
        return -1;
    }
    else {
        // Файл успешно открыт, можно читать из него данные
        return 1;
    }
    file.close(); // Закрытие файла

    return 0;
}

```

Как показано в примере выше, для проверки успешности открытия файла, можно использовать метод `is_open()`. Этот метод возвращает `true`, если файл успешно открыт, и `false` в противном случае. Также можно использовать логическое

выражение с оператором "!", чтобы проверить, не равна ли переменная файла нулю. Также после работы с файлом его нужно закрыть методом close().

Чтение из файла

Существует несколько методов чтения данных из файла:

1. Метод getline() - позволяет считать строку из файла до символа новой строки или до конца файла. Пример:

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

int main() {
    setlocale(LC_ALL, "rus");
    ifstream file("C:/file.txt");
    if (!file.is_open()) {
        cerr << "Файл не может быть открыт!";
        return 1;
    }

    string line;
    while (getline(file, line)) {
        cout << line;
    }

    file.close();

    return 0;
}
```

2. Оператор >> - позволяет считывать данные из файла по одному слову или числу. Пример:

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

int main() {
    setlocale(LC_ALL, "rus");
    ifstream file("C:/file.txt");
    if (!file.is_open()) {
        cerr << "Файл не может быть открыт!";
    }
}
```

```

    return 1;
}

int number;
while (file >> number) {
    cout << number;
}

file.close();

return 0;
}

```

3. Метод `get()` - позволяет считывать данные из файла по одному символу.

```

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

int main() {
    setlocale(LC_ALL, "rus");
    ifstream file("C:/file.txt");
    if (!file.is_open()) {
        cerr << "Файл не может быть открыт!\n";
    }

    return 1;
}

char symbol;
while (file.get(symbol)) {
    cout << symbol;
}

file.close();

return 0;
}

```

4. Метод `read()` - позволяет считать данные из файла в буфер заданного размера. Пример:

```

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

int main() {
    setlocale(LC_ALL, "rus");
    ifstream file("C:/file.txt", ios::binary);
    if (!file.is_open()) {
        cerr << "Файл не может быть открыт!";

        return 1;
    }
    char buffer[256];
    while (file.read(buffer, sizeof(buffer))) {
        cout.write(buffer, file.gcount());
    }
    file.close();

    return 0;
}

```

Обратите внимание, что для чтения файлов в бинарном режиме необходимо указать флаг `ios::binary` при открытии файла.

Запись в файл

Когда вы работаете с файлами в C++, вы можете записывать данные в файлы с помощью объектов типа `ofstream`. В C++ есть несколько методов, которые можно использовать для записи данных в файлы. Вот некоторые из них:

1. Метод `<<` оператора: Этот метод записывает данные в файл с помощью перегруженного оператора `<<`. Этот метод может использоваться для записи простых типов данных, таких как `int`, `double` и `char`, а также пользовательских типов данных, которые были перегружены для работы с оператором `<<`.

```

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

int main() {
    ofstream myfile;
    myfile.open("file.txt");
    myfile << "Вывод в файл";
    myfile << 123 << endl;
    myfile.close();
}

```

```
    return 0;
}
```

2. Метод write(): Этот метод записывает данные в файл, используя указатель на массив байтов и количество байтов для записи. Этот метод может использоваться для записи произвольных данных в файл. Вот пример кода:

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

int main() {
    ofstream myfile;
    myfile.open("file.bin", ios::binary);
    int data[] = { 1, 2, 3, 4, 5 };
    int size = sizeof(data);
    myfile.write((char*)&data, size);
    myfile.close();

    return 0;
}
```

Этот код открывает файл "file.bin" для записи бинарных данных и записывает в него массив целых чисел.

3. Метод put(): Этот метод записывает один символ в файл. Этот метод может использоваться для записи символов в файл. Вот пример кода:

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

int main() {
    ofstream myfile;
    myfile.open("file.txt");
    myfile.put('H');
    myfile.put('i');
    myfile.close();
    return 0;
}
```

Методы форматирования width, precision

Конструкция `width()` используется для указания минимальной ширины выводимого значения. Например, чтобы выровнять числа в колонке таблицы. Конструкция `precision()` используется для указания количества знаков после запятой, которые должны быть выведены для чисел с плавающей точкой. Рассмотрим пример:

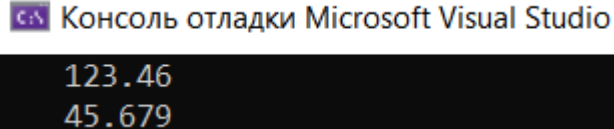
```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    double num1 = 123.456789;
    double num2 = 45.6789123;

    cout << setw(10) << setprecision(5) << num1 << endl;
    cout << setw(10) << setprecision(5) << num2 << endl;

    return 0;
}
```

В этом примере мы используем `setw(10)`, чтобы установить минимальную ширину поля в 10 символов. Мы также используем `setprecision(3)`, чтобы указать, что мы хотим вывести только три знака.



Консоль отладки Microsoft Visual Studio

```
123.46
45.679
```

Режимы открытия файлов

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

Константа	Описание
<code>ios_base::in</code>	открыть файл для чтения
<code>ios_base::out</code>	открыть файл для записи
<code>ios_base::ate</code>	при открытии переместить указатель в конец файла

ios_base::app	открыть файл для записи в конец файла
ios_base::trunc	удалить содержимое файла, если он существует
ios_base::binary	открытие файла в двоичном режиме

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

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

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

Объекты класса `ofstream`, при связке с файлами по умолчанию содержат режимы открытия файлов `ios_base::out | ios_base::trunc`. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи. Объекты класса `ifstream` связываясь с файлом, имеют по умолчанию режим открытия файла `ios_base::in` — файл открыт только для чтения.

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

Ознакомьтесь с примером применения вышеперечисленных методов для записи и чтения структуры из файла.

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

struct Person { // Определяем структуру
    string name;
    int age;
};

int main() {
    ofstream outFile("persons.txt"); // Создаем объект типа ofstream для записи в файл
    if (!outFile) { // Проверяем, успешно ли открыт файл для записи
        cerr << "Не удалось открыть файл для записи!" << endl;
    }
}
```



```

    return 1;
}
Person person; // Создаем объект типа Person
person.name = "Соня";
person.age = 19;

outFile << person.name << " " << person.age << endl; // Записываем структуру в
файл

outFile.close(); // Закрываем файл
ifstream inFile("persons.txt"); // Создаем объект типа ifstream для чтения из файла
if (!inFile) { // Проверяем, успешно ли открыт файл для чтения

    cerr << "Не удалось открыть файл для чтения!" << endl;
    return 1;
}

Person readPerson; // Читаем структуру из файла
inFile >> readPerson.name;
inFile >> readPerson.age;

cout << "Имя: " << readPerson.name << endl; // Выводим прочитанную структуру
на экран
cout << "Возраст: " << readPerson.age << endl;

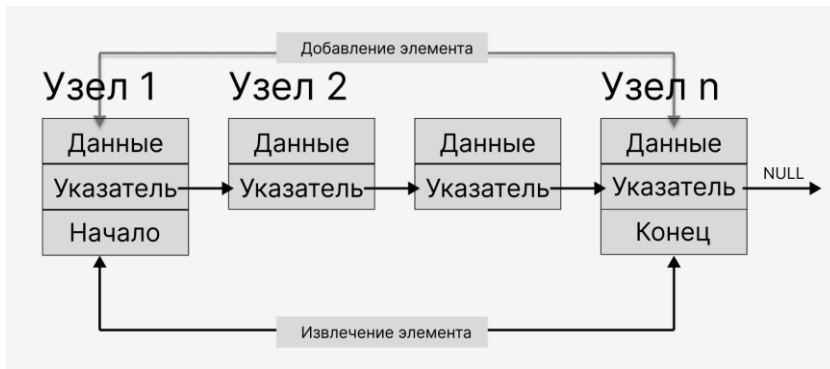
inFile.close(); // Закрываем файл
return 0;
}

```

Теоретические сведения № 2

Двунаправленные очереди — это абстрактный тип данных, который можно рассматривать как список элементов, доступных для чтения и записи с обоих концов (то есть он поддерживает как стековые, так и очередные операции).

Это означает, что вы можете добавлять и удалять элементы как в начале, так и в конце очереди.



Для его использования необходимо подключить заголовочный файл `<deque>`.
Способы создания двухсторонней очереди:

```
deque<int> deque1; // пустая очередь
deque<int> deque2(5); // deque2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию
deque<int> deque(5, 2); // deque3 состоит из 5 чисел, каждое число равно 2
deque<int> deque4{ 1, 2, 4, 5 }; // deque4 состоит из чисел 1, 2, 4, 5
deque<int> deque5 = { 1, 2, 3, 5 }; // deque5 состоит из чисел 1, 2, 3, 5
deque<int> deque6({ 1, 2, 3, 4, 5 }); // deque6 состоит из чисел 1, 2, 3, 4, 5
deque<int> deque7(deque4); // deque7 - копия очереди deque4
deque<int> deque8 = deque7; // deque8 - копия очереди deque7
```

Для получения элементов очереди можно использовать ряд функций:

- **[index]**: получение элемента по индексу
- **at(index)**: возвращает элемент по индексу
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<int> numbers{ 1, 2, 3, 4, 5 };
    int first = numbers.front(); // 1
    int last = numbers.back(); // 5
    int second = numbers[1]; // 2
    int third = numbers.at(2); // 3
    cout << first << second << third << last << endl; // 1235
}
```

Чтобы узнать размер очереди, можно использовать функцию **size()**. А функция **empty()** позволяет узнать, содержит ли очередь элементы. Она возвращает значение **true**, если в очереди есть элементы:

```
#include <iostream>
#include <deque>

using namespace std;

int main(){
    deque<int> numbers{ 1, 2, 3, 4, 5 };
    if (numbers.empty())
    {
        cout << "Deque is empty" << endl;
    }
    else
    {
        cout << "Deque has " << numbers.size() << " elements" << endl;// 5
    }
}
```

Функция **resize()** позволяет изменить размер очереди. Эта функция имеет две формы:

- **resize(n)**: оставляет в очереди **n** первых элементов. Если **deque** содержит больше элементов, то размер контейнера усекается до первых **n** элементов. Если размер очереди меньше **n**, то добавляются недостающие элементы и инициализируются значением по умолчанию (0)
- **resize(n, value)**: также оставляет в очереди **n** первых элементов. Если размер очереди меньше **n**, то добавляются недостающие элементы со значением **value**

```
#include <iostream>
#include <deque>

using namespace std;

int main(){
    deque<int> numbers{ 1, 2, 3, 4, 5 };
    numbers.resize(4); // оставляем первые четыре элемента - numbers = {1, 2, 3, 4}
    cout << "Deque size is " << numbers.size() << endl;// 4
    numbers.resize(7, 8); // numbers = {1, 2, 3, 4, 8, 8, 8}
    cout << "Deque size is " << numbers.size() << endl;// 7
}
```

Функция **assign()** позволяет заменить все элементы очереди определенным набором. Она имеет следующие формы:

- **assign(il)**: заменяет содержимое контейнера элементами из списка инициализации **il**

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
numbers.assign({ 21, 22 });
// numbers = { 21, 22 }
```

- **assign(n, value)**: заменяет содержимое контейнера **n** элементами, которые имеют значение **value**

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
numbers.assign(4, 3);
// numbers = { 3, 3, 3, 3 }
```

- **assign(begin, end)**: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы **begin** и **end**

```
deque<int> values{ 6, 7, 8, 9, 10, 11 };
auto start = values.begin() + 2; // итератор указывает на третий элемент
auto end = values.end(); // итератор указывает на последний элемент
numbers.assign(start, end);
// numbers = { 8, 9, 10, 11 }
```

Функция **swap()** обменивает значениями две очереди:

```
deque<int> deque1{ 1, 2, 3, 4, 5 };
deque<int> deque2{ 6, 7, 8, 9 };
deque1.swap(deque2);
// deque1 = { 6, 7, 8, 9 };
// deque2 = { 1, 2, 3, 4, 5 };
```

Чтобы добавить элементы в очередь **deque**, можно применять ряд функций:

- **push_back(val)**: добавляет значение **val** в конец очереди
- **push_front(val)**: добавляет значение **val** в начало очереди
- **emplace_back(val)**: добавляет значение **val** в конец очереди. Эта функция является более эффективной, чем **push_back()**, когда мы создаем новый элемент, потому что она создает его прямо в памяти очереди, вместо того, чтобы создавать временный объект и потом копировать его в очередь.

- **emplace_front(val)**: добавляет значение val в начало очереди. Она также является более эффективной, чем push_front(), если мы создаем новый элемент, потому что он создает его прямо в памяти очереди.

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
numbers.push_back(6); // { 1, 2, 3, 4, 5, 6 }
numbers.push_front(0); // { 0, 1, 2, 3, 4, 5, 6 }
numbers.emplace_back(7); // { 0, 1, 2, 3, 4, 5, 6, 7 }
numbers.emplace_front(-1); // { -1, 0, 1, 2, 3, 4, 5, 6, 7 }
```

- **emplace(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = ++numbers.cbegin(); // итератор указывает на второй элемент
numbers.emplace(iter, 8); // { 1, 8, 2, 3, 4, 5 };
```

- **insert(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // итератор указывает на второй элемент
numbers.insert(iter + 2, 8); // добавляем после второго элемента
//numbers = { 1, 2, 8, 3, 4, 5 };
```

- **insert(pos, n, val)**: вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.

```
deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // итератор указывает на первый элемент
numbers.insert(iter, 3, 4); // добавляем вначале три четверки
//numbers = { 4, 4, 4, 1, 2, 3, 4, 5 };
```

- **insert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.

```

    deque<int> values{ 10, 20, 30, 40, 50 };
deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // итератор указывает на первый элемент
// добавляем в начало все элементы из values
numbers.insert(iter, values.begin(), values.end());
//numbers = { 10, 20, 30, 40, 50, 1, 2, 3, 4, 5 };

```

- **insert(pos, values):** вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

```

    deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cend(); // итератор указывает на позицию за последним
элементом
// добавляем после последнего элемента список { 21, 22, 23 }
numbers.insert(iter, { 21, 22, 23 });
//numbers = { 1, 2, 3, 4, 5, 21, 22, 23 };

```

Для удаления элементов из контейнера deque используются следующие функции:

- **clear(p):** удаляет все элементы
- **pop_back():** удаляет последний элемент
- **pop_front():** удаляет первый элемент

```

    deque<int> numbers{ 1, 2, 3, 4, 5 };
numbers.pop_front(); // numbers = { 2, 3, 4, 5 }
numbers.pop_back(); // numbers = { 2, 3, 4 }
numbers.clear(); // numbers = {}

```

- **erase(p):** удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент

```

    deque<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // указатель на первый элемент
numbers.erase(iter); // удаляем первый элемент
// numbers = { 2, 4, 5, 6 }

```

- **erase(begin, end):** удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент,

следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

```
deque<int> numbers{ 1, 2, 3, 4, 5 };  
auto begin = numbers.begin(); // указатель на первый элемент  
auto end = numbers.end();      // указатель на последний элемент  
numbers.erase(++begin, --end); // удаляем со второго элемента до последнего  
//numbers = { 1, 5 }
```

Задания к лабораторной работе №4

Задание 1.

Реализовать возможность записи данных из структуры вашей бд в файл, а также вывод содержимого файла.

Задание 2.

Написать функцию для считывания данных из файла в структуру вашей БД.

Задание 3.

Написать программу, в которой нужно найти минимальный элемент деки и добавить его в начало, с выводом на экран .

Задание 4.

Написать программу, в которой нужно найти произведение всех элементов деки и вывести его на экран.

Контрольные вопросы

1. Что такое файл?
1. В чём отличие между текстовым и бинарным файлом?
2. Что такое потоки `ifstream` и `ofstream`?
3. Какими способами можно считывать данные из файла?
4. Какими способами можно записывать данные в файл?
5. Какие режимы открытия файлов бывают и в чём их отличие?
6. Что такое `deque` в C++?
7. Как объявить и инициализировать `deque` в C++?
8. Как добавить элемент в начало и конец `deque` в C++?
9. Как удалить элемент из начала и конца `deque` в C++?
10. Как получить доступ к элементу `deque` по индексу в C++?
11. Как получить размер `deque` в C++?
12. Как проверить, пустой ли `deque` в C++?
13. Как выполнить поиск элемента в `deque` в C++?

Дополнительные задания

Дополнительное задание 1.

Для вашей третьей структуры реализуйте ввод и вывод данных в двоичный файл.

Дополнительное задание 2.

Напишите программу, которая запрашивает у пользователя последовательность целых чисел и сохраняет их в контейнере `deque`. Затем программа должна отсортировать эту последовательность в порядке возрастания и вывести на экран первые 5 элементов.

Дополнительное задание 3.

«Злые птицы». Есть провод конечной длины, на котором располагаются птицы. Некоторые птицы идут направо, некоторые — налево. Когда встречаются друг с другом — меняют направление. Когда доходят до конца провода — взлетают. Нужно определить, когда взлетит каждая из этих птиц.

Вам необходимо написать программу, которая по длине провода, начальным позициям и направлениям бега птиц выяснит для каждой птицы, в какой момент времени она улетит с провода.

Формат входных данных:

В первой строке задано единственное целое число L ($1 \leq L \leq 109$) — длина провода в метрах. Во второй строке записано число n ($0 \leq n \leq 100\,000$) — количество птиц, бегущих направо. В третьей строке записано n различных целых чисел a_i ($0 < a_i < L$) — расстояния в метрах от левого конца провода до птиц, бегущих направо. В четвертой строке записано число m ($0 \leq m \leq 100\,000$) — количество птиц, бегущих налево. В пятой строке записано m различных целых чисел b_i ($0 < b_i < L$) — расстояния в метрах от левого конца провода до птиц, бегущих налево. Никакие две птицы не находятся исходно в одном и том же месте. Гарантируется, что на проводе сидит хотя бы одна птица.

Формат выходных данных

В первой строке выведите n целых чисел t_i — через сколько минут улетит i -я по порядку описания во вводе птица, бегущая направо. Во второй строке выведите m целых чисел u_i — через сколько минут улетит i -я по порядку описания во вводе птица, бегущая налево.

Лабораторная работа № 5. Редактирование файлов: частичное и полное удаление, изменение поля. Редактирование массива структур. Forward_list

Цель: изучить способы частичного и полного удаления данных из файла; научиться редактировать массив структур, научиться работать с контейнером `<forward_list>`

Теоретические сведения № 1

В C++ существует несколько способов удаления данных из файла. Некоторые из них перечислены ниже:

1. Открытие файла в режиме `ios::trunc`

Как упоминалось ранее, при открытии файла в режиме `ios::trunc` содержимое файла обрезается до нулевой длины. Если файл не существует, он создается.

```
ofstream myfile;  
myfile.open("example.txt", ios::out | ios::trunc);
```

2. Использование функции `remove()`

Функция `remove()` удаляет файл из файловой системы. Она принимает в качестве аргумента имя файла, который нужно удалить.

```
#include <cstdio>  
  
int main() {  
    if (remove("example.txt") != 0) {  
        perror("Ошибка удаления файла");  
    }  
    else {  
        puts("Файл успешно удален");  
    }  
    return 0;  
}
```

Чтобы удалить структуру из массива структур по индексу, вы можете использовать следующий алгоритм:

1. Сначала скопируйте все структуры, которые находятся после удаляемой структуры, на одну позицию влево. Это можно сделать с помощью цикла `for`.

2. Затем уменьшите размер массива на 1, чтобы удалить последнюю структуру, которую вы только что скопировали на предыдущем шаге.

Пример кода, который демонстрирует, как удалить структуру из массива структур по индексу:

```

#include <iostream>
using namespace std;

struct Person {
    string name;
    int age;
};

int main() {
    Person people[3] = { {"Alice", 25}, {"Bob", 30}, {"Charlie", 35} };
    int index = 1; // Индекс удаляемой структуры

    // Создание нового массива меньшего размера
    Person newPeople[2];

    // Копирование всех структур до удаляемой структуры
    for (int i = 0; i < index; i++) {
        newPeople[i] = people[i];
    }

    // Копирование всех структур после удаляемой структуры на одну позицию
    влево
    for (int i = index; i < 2; i++) {
        newPeople[i] = people[i + 1];
    }

    // Вывод оставшихся структур
    for (int i = 0; i < 2; i++) {
        cout << newPeople[i].name << " " << newPeople[i].age << endl;
    }
    return 0;
}

```

После удаления одной из структур из массива мы можем переписать файл, чтоб у нас хранилась обновленная информация.

Далее разберём изменение структуры в массиве по индексу, проверяя, находится ли индекс в пределах массива. Если индекс находится в пределах массива, мы изменяем поля name и age структуры по этому индексу. Если же индекс находится за пределами массива, мы выводим сообщение об ошибке. Затем мы выводим все структуры в массиве, чтобы убедиться в том, что изменения были выполнены правильно.

```

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

struct Person {
    string name;
    int age;
};

int main() {
    Person people[3] = { {"Alice", 25}, {"Bob", 30}, {"Charlie", 35} };

    // Редактирование структуры по индексу
    int index = 1;
    if (index >= 0 && index < 3) {
        people[index].name = "Bill";
        people[index].age = 28;
    }

    else {
        cout << "Индекс за пределами массива" << endl;
    }

    // Вывод отредактированных структур
    for (int i = 0; i < 3; i++) {
        cout << people[i].name << " " << people[i].age << endl;
    }

    return 0;
}

```

Теоретические сведения № 2

Контейнер `forward_list` представляет односвязный список.

Односвязный список - это динамическая структура данных, состоящая из узлов. Каждый узел будет иметь какое-то значение и указатель на следующий узел.



Для использования данного типа списка необходимо подключить заголовочный файл `forward_list`.

```
#include <forward_list>
```

Способы создания односвязного списка:

```
std::forward_list<int> list1;           // пустой список
std::forward_list<int> list2(5, 2);     // list2 состоит из 5 чисел, каждое число
равно 2
std::forward_list<int> list3{ 1, 2, 4, 5 }; // list3 состоит из чисел 1, 2, 4, 5
std::forward_list<int> list4 = { 1, 2, 3, 4, 5 }; // list4 состоит из чисел 1, 2, 3, 4, 5
std::forward_list<int> list5(list4);    // list5 - копия списка list4
std::forward_list<int> list6 = list4;   // list7 - копия списка list4
```

Напрямую в списке `forward_list` можно получить только первый элемент. Для этого применяется функция **front()**

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

using namespace std;

int main()
{
    forward_list<int> numbers{ 1, 2, 3, 4, 5 };
    int first{ numbers.front() };

    cout << "First: " << first << endl; // 1
    for (int n : numbers)
        cout << n << "\t"; // 1 2 3 4 5
    cout << endl;
}
```

По умолчанию класс `forward_list` не определяет никаких функций, которые позволяют получить размер контейнера. В этом классе есть только функция **max_size()**, которая позволяет получить максимально доступный размер контейнера.

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

using namespace std;

int main()
{
    forward_list<int> numbers{ 1, 2, 3, 4, 5 };
    cout << "Max size: " << numbers.max_size() << endl; // 1152921504606846975
}
```

```

forward_list<int> numbers2{ 6, 7, 8 };
cout << "Max size: " << numbers2.max_size() << endl; // 1152921504606846975
}

```

Таким образом максимальный размер контейнера является неизменным.

Для изменения размера контейнера можно использовать функцию **resize()**, которая имеет две формы:

- **resize(n)**: оставляет в списке n первых элементов. Если список содержит больше элементов, то он усекается до первых n элементов. Если размер списка меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию
- **resize(n, value)**: также оставляет в списке n первых элементов. Если размер списка меньше n, то добавляются недостающие элементы со значением value

```

#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
    forward_list<int> numbers{ 1, 2, 3, 4, 5, 6 };
    cout << "Forward list:" << endl;
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
    cout << "First change:" << endl;
    numbers.resize(4); // оставляем первые четыре элемента
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
    cout << "Second change:" << endl;
    numbers.resize(6, 8); // указываем размер списка как 6, заполняем пустые
    значения 8-ой
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
}

```

Функция **assign()** позволяет заменить все элементы списка определенным набором. Она имеет следующие формы:

- **assign(il)**: заменяет содержимое контейнера элементами из списка инициализации il

- **assign(n, value)**: заменяет содержимое контейнера n элементами, которые имеют значение value
- **assign(begin, end)**: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы begin и end

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

using namespace std;

int main()
{
    forward_list<int> numbers{ 1, 2, 3, 4, 5 };
    for (int n : numbers)//1 2 3 4 5
        cout << n << "\t";
    cout << endl;
    numbers.assign({ 21, 22, 23, 24, 25 });
    for (int n : numbers)//21 22 23 24 25
        cout << n << "\t";
    cout << endl;
    numbers.assign(4, 3);
    for (int n : numbers)//3 3 3 3
        cout << n << "\t";
    cout << endl;
    forward_list<int> values{ 6, 7, 8, 9, 10, 11 };
    auto start = ++values.begin(); // итератор указывает на второй элемент из
values
    auto end = values.end();
    numbers.assign(start, end);
    for (int n : numbers)//7 8 9 10 11
        cout << n << "\t";
    cout << endl;
}
```

Функция **swap()** обменивает значениями два списка:

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

using namespace std;

int main()
{
    forward_list<int> numbers1{ 1, 2, 3, 4, 5 };
    cout << "First list:" << endl;//1 2 3 4 5
```

```

for (int n1 : numbers1)
    cout << n1 << "\t";
cout << endl;
forward_list<int> numbers2{ 6, 7, 8, 9 };
cout << "Second list:" << endl; //6 7 8 9
for (int n2 : numbers2)
    cout << n2 << "\t";
cout << endl;
numbers1.swap(numbers2);
cout << "First list : " << endl; //6 7 8 9
for (int n1 : numbers1)
    cout << n1 << "\t";
cout << endl;
cout << "Second list" << endl; //1 2 3 4 5
for (int n2 : numbers2)
    cout << n2 << "\t";
cout << endl;
}

```

Для добавления элементов в `forward_list` применяются следующие функции:

1. **`push_front(val)`**: добавляет объект `val` в начало списка
2. **`emplace_front(val)`**: добавляет объект `val` в начало списка

Разница между функциями `push_front()` и `emplace_front()` заключается в том, каким образом они добавляют новый элемент в очередь.

- `push_front(val)`: добавляет значение `val` в начало очереди, реализуя операцию "вставки в начало" очереди.

- `emplace_front(val)`: добавляет новый элемент со значением `val` в начало очереди, используя конструктор класса. Она также является более эффективной, чем `push_front()`, если мы создаем новый элемент, потому что он создает его прямо в памяти очереди.

В целом, разница между этими функциями заключается в том, как они добавляют элементы в очередь. Выбор той или иной функции зависит от типа добавляемых элементов и их количества, а также от того, успевают ли мы создать новый объект перед вставкой, и какой операции вставки требуется в текущем контексте.

3. **`emplace_after(p, val)`**: вставляет объект `val` после элемента, на который указывает итератор `p`. Возвращает итератор на вставленный элемент. Если `p` представляет итератор на позицию после конца списка, то результат неопределен.

4. **`insert_after(p, val)`**: вставляет объект `val` после элемента, на который указывает итератор `p`. Возвращает итератор на вставленный элемент.

Обе функции `emplace_after(p, val)` и `insert_after(p, val)` используются для вставки нового элемента в односвязный список и добавления его после элемента, на который указывает итератор `p`. Тем не менее, различия между ними заключаются в следующем:

- `emplace_after(p, val)` позволяет создавать и вставлять новый элемент с использованием конструктора класса, который оптимально использует память. Она возвращает итератор на только что вставленный элемент.

- `insert_after(p, val)` вставляет копию элемента `val` в односвязный список. Она также возвращает итератор на только что вставленный элемент.

Это означает, что `emplace_after()` может быть более производительным, поскольку он создает новый элемент прямо в памяти списка и не требует создания временного объекта, который копируется в список.

5. `insert_after(p, n, val)`: вставляет `n` объектов `val` после элемента, на который указывает итератор `p`. Возвращает итератор на последний вставленный элемент.

6. `insert_after(p, begin, end)`: вставляет после элемента, на который указывает итератор `p`, набор объектов из другого контейнера, начало и конец которого определяется итераторами `begin` и `end`. Возвращает итератор на последний вставленный элемент.

7. `insert_after(p, il)`: вставляет после элемента, на который указывает итератор `p`, список инициализации `il`. Возвращает итератор на последний вставленный элемент.

Используем некоторые функции в примере.

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

using namespace std;

int main()
{
    forward_list<int> numbers{ 2, 3, 4 };
    numbers.push_front(1);
    // numbers = { 1, 2, 3, 4 }
    numbers.emplace_front(0); // добавляем в начало число 0
    // numbers = { 0, 1, 2, 3, 4 }
    auto iter = numbers.begin();
    iter = numbers.emplace_after(iter, 0); // добавляем после итератора число 0
    // numbers = { 0, 0, 1, 2, 3, 4 }
    for (int n : numbers) cout << n << " ";
}
```



```
cout << endl;
}
```

Чтобы удалить элемент из контейнера `forward_list`, можно использовать следующие функции:

- **clear()**: удаляет все элементы
- **pop_front()**: удаляет первый элемент
- **erase_after(p)**: удаляет элемент после элемента, на который указывает итератор `p`. Возвращает итератор на элемент после удаленного
- **erase_after(begin, end)**: удаляет диапазон элементов, на начало и конец которого указывают соответственно итераторы `begin` и `end`. Возвращает итератор на элемент после последнего удаленного

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

using namespace std;

int main()
{
    std::forward_list<int> numbers{ 1, 2, 3, 4, 5 };
    numbers.pop_front();
    // numbers = { 2, 3, 4, 5 };
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
    auto iter = numbers.erase_after(numbers.begin());
    // numbers = { 2, 4, 5 };
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
    numbers.erase_after(iter, numbers.end());
    // numbers = { 2, 4 };
    for (int n : numbers)
        cout << n << "\t";
    cout << endl;
}
```

Задания к лабораторной работе №5

Задание 1.

Написать функции удаления(очистки) содержимого вашей БД и полного удаления вашей БД. Функции удаляют информацию как из файла, так и из структуры БД.

Задание 2.

Написать функции удаления и редактирования элемента структуры по индексу в вашей БД.

Задание 3.

Написать программу, в которой можно осуществить вставку n-новых, после введенного пользователем эл-та в односвязный список, с выводом на экран.

Задание 4.

Написать программу, в которой нужно удалить все элементы односвязного списка и добавить n новых, с выводом на экран.

Контрольные вопросы

1. Какие есть способы удаления данных из файла?
2. Как можно удалить экземпляр структуры?
3. Как можно удалить экземпляр структуры по индексу?
4. Как можно отредактировать экземпляр структуры по индексу?
5. Что такое `forward_list` в C++?
6. Как объявить и инициализировать `forward_list` в C++?
7. Как добавить элемент в начало и конец `forward_list` в C++?
8. Как удалить элемент из начала и конца `forward_list` в C++?
9. Как получить доступ к элементу `forward_list` по индексу в C++?
10. Как получить размер `forward_list` в C++?
11. Как выполнить поиск элемента в `forward_list` в C++?
- 12.

Дополнительные задания

Дополнительное задание 1.

Создайте текстовый файл с данными. Напишите программу, которая будет считывать данные из этого файла и удалять из него записи, содержащие определенное ключевое слово (например, "удалить"). При удалении записи программа должна выводить на экран информацию о том, какая запись была удалена и сколько записей осталось в файле.

Дополнительное задание 2.

Напишите программу, которая запрашивает у пользователя список слов и сохраняет их в контейнере `forward_list` в STL C++. Затем программа должна удалить все слова, которые начинаются на гласную букву, и вывести на экран оставшиеся слова. Обратите внимание, что программа должна игнорировать регистр букв при определении гласных букв.

Дополнительное задание 3.

Написать программу, которая принимает на вход два связанных списка (`forward_list`) и объединяет их в один связный список. Результатом должен быть

новый связный список, содержащий все элементы из двух исходных списков, отсортированных по возрастанию.

Лабораторная работа № 6. Сортировки. List

Цель:изучить методы сортировок и использовать их на практике, научиться работать с контейнером <list>

Теоретические сведения № 1

Сортировка - это процесс упорядочивания элементов в массиве или контейнере в соответствии с определенным критерием. Существует множество алгоритмов сортировки, каждый из которых имеет свои преимущества и недостатки.

Одной из основных причин использования сортировок является необходимость упорядочивания данных для более эффективного поиска и обработки. Например, если вы хотите найти определенный элемент в массиве, то поиск будет происходить гораздо быстрее, если элементы в массиве будут отсортированы по возрастанию или убыванию.

Сортировки также могут использоваться для оптимизации алгоритмов, которые работают с отсортированными данными. Например, бинарный поиск, который ищет элементы в отсортированном массиве, работает значительно быстрее, чем линейный поиск, который перебирает все элементы.

Кроме того, сортировки могут быть полезны для визуализации и анализа данных. Например, если вы хотите проанализировать распределение данных в массиве, то сортировка может помочь визуализировать эти данные в виде гистограммы или диаграммы.

Сортировки являются важным инструментом в программировании и могут помочь ускорить поиск и обработку данных, улучшить производительность алгоритмов и упростить анализ данных.

Сортировка пузырьком

Алгоритм сортировки пузырьком (Bubble Sort) - это простой алгоритм сортировки, который работает путем сравнения и обмена соседних элементов в массиве. Алгоритм получил свое название из-за того, что большие элементы "всплывают" вверх массива, как пузырьки воды, во время выполнения сортировки.

Алгоритм сортировки пузырьком можно реализовать в нескольких вариантах, но основная идея остается неизменной. Вот пример реализации сортировки пузырьком:

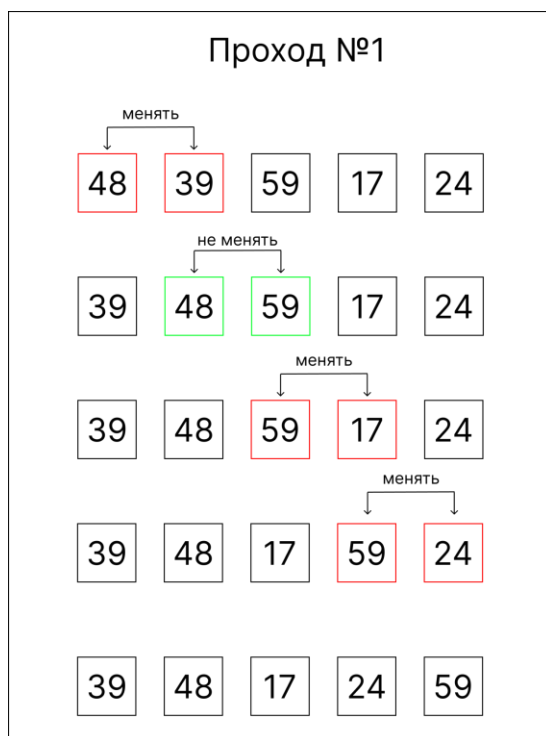
```
void bubbleSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
}  
}
```

В этом примере используются два вложенных цикла for для прохода по всем элементам массива и сравнения каждого элемента со своим соседом. Если элемент слева больше элемента справа, то мы меняем их местами.

Алгоритм сортировки пузырьком имеет сложность $O(N^2)$, что делает его неэффективным для сортировки больших массивов. Однако, этот алгоритм все еще может быть полезным для сортировки небольших массивов или для обучения основам алгоритмов сортировки.

Алгоритм сортировки пузырьком - это простой и интуитивно понятный алгоритм сортировки, который может быть легко реализован на C++. Однако, из-за его низкой эффективности, в большинстве случаев лучше использовать более продвинутые алгоритмы сортировки, такие как быстрая сортировка или сортировка слиянием.





Сортировка вставками

Алгоритм сортировки вставками (insertion sort) работает по следующему принципу: на каждом шаге мы берем очередной элемент массива и вставляем его в отсортированную часть массива так, чтобы она оставалась отсортированной. Таким образом, мы постепенно увеличиваем отсортированную часть массива до тех пор, пока не отсортируем весь массив.

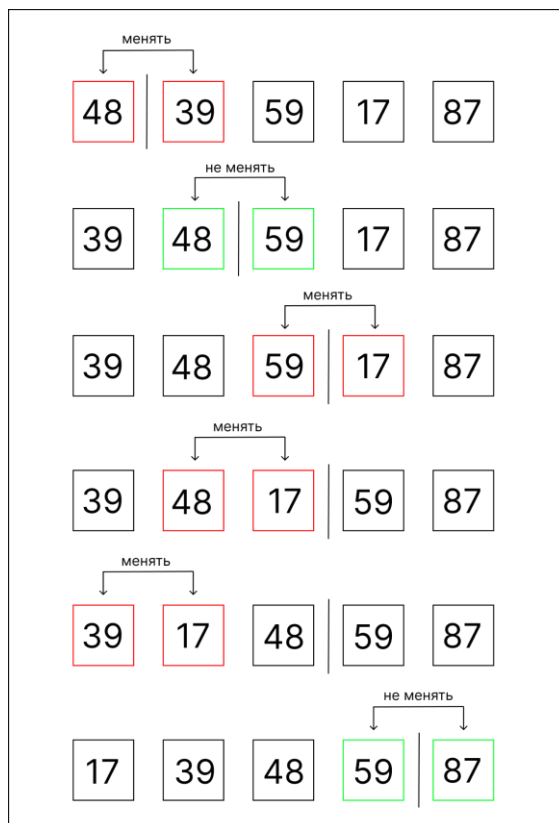
```

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

```

В этом коде мы проходим по массиву начиная с первого элемента $i = 1$. На каждом шаге мы берем текущий элемент `key`, и сравниваем его со всеми элементами, которые идут перед ним, начиная с элемента $j = i - 1$. Если мы находим элемент, который больше `key`, то мы сдвигаем его на одну позицию вправо, чтобы освободить место для `key`. Мы продолжаем сдвигать элементы до тех пор, пока не найдем место, куда можно вставить `key`. Затем мы вставляем `key` в эту позицию.

Сортировка вставками имеет сложность $O(n^2)$, что делает ее не очень эффективной для больших массивов. Однако, она имеет некоторые преимущества перед другими алгоритмами сортировки, например, она быстрее работает на уже почти отсортированных массивах.



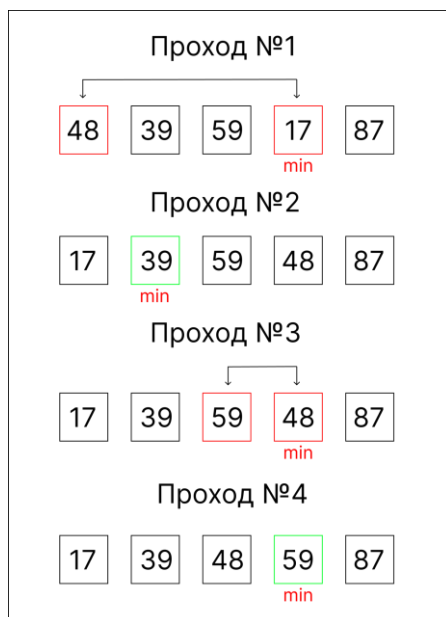
Сортировка выбором

Алгоритм сортировки выбором (selection sort) работает по следующему принципу: на каждом шаге мы ищем минимальный элемент в неотсортированной части массива и меняем его местами с первым элементом в неотсортированной части. Таким образом, мы постепенно увеличиваем отсортированную часть массива до тех пор, пока не отсортируем весь массив.

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

В этом коде мы проходим по массиву начиная с первого элемента $i = 0$. На каждом шаге мы ищем минимальный элемент в неотсортированной части массива, начиная со следующего элемента после i , и меняем его местами с первым элементом в неотсортированной части, который имеет индекс i . Мы продолжаем так делать до тех пор, пока не отсортируем весь массив.

Сортировка выбором также имеет сложность $O(n^2)$, что делает ее не очень эффективной для больших массивов. Однако, она также имеет некоторые преимущества перед другими алгоритмами сортировки, например, она менее затратна по памяти, так как не использует дополнительную память для сортировки.



Сортировка слиянием

Алгоритм сортировки слиянием (merge sort) работает по следующему принципу: мы делим исходный массив на две части, рекурсивно сортируем каждую из них и затем сливаем две отсортированные части в один отсортированный массив. Таким образом, мы постепенно уменьшаем размеры частей до тех пор, пока не получим отсортированные отдельные элементы, которые затем сливаем в один отсортированный массив.

```
void merge(int arr[], int left, int middle, int right) {
    int size_left = middle - left + 1;
    int size_right = right - middle;
    int* left_arr = new int[size_left];
    int* right_arr = new int[size_right];

    for (int i = 0; i < size_left; i++) {
        left_arr[i] = arr[left + i];
    }
    for (int j = 0; j < size_right; j++) {
        right_arr[j] = arr[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < size_left && j < size_right) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];
            i++;
        }
        else {
            arr[k] = right_arr[j];
            j++;
        }
        k++;
    }

    while (i < size_left) {
        arr[k] = left_arr[i];
        i++;
        k++;
    }

    while (j < size_right) {
        arr[k] = right_arr[j];
        j++;
        k++;
    }
}
```

```

    }

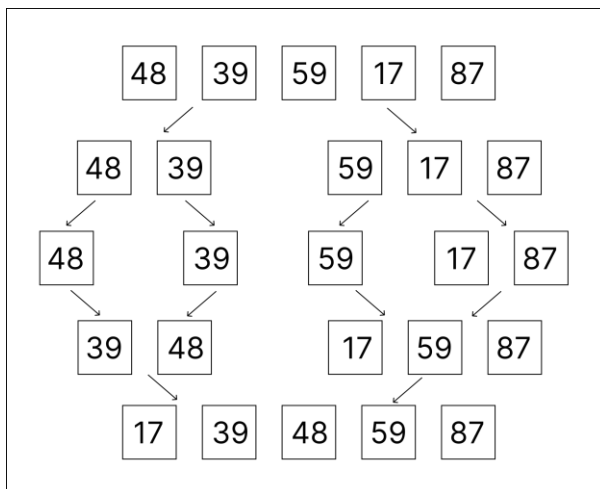
    delete[] left_arr;
    delete[] right_arr;
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

```

В этом коде мы используем функцию `merge()`, которая сливает две отсортированные части массива (от `l` до `m` и от `m+1` до `r`) в один отсортированный массив. Затем мы используем функцию `mergeSort()`, которая рекурсивно разделяет исходный массив на две части и сортирует каждую из них с помощью `merge()`. Мы продолжаем делать это до тех пор, пока не получим отдельные отсортированные элементы, которые затем сливаем в один отсортированный массив.

Сортировка слиянием имеет сложность $O(n \log n)$, что делает ее более эффективной для больших массивов, чем сортировки со сложностью $O(n^2)$. Однако, она требует дополнительной памяти для слияния частей массива, что может быть проблемой для очень больших массивов.



Быстрая сортировка

Алгоритм быстрой сортировки (quick sort) основан на методе "разделяй и властвуй". Он работает по следующему принципу: мы выбираем опорный элемент из массива, затем перемещаем все элементы, меньшие опорного, на его левую сторону, а все элементы, большие опорного, на правую сторону. Таким образом, мы разбиваем массив на две части и рекурсивно повторяем этот процесс для каждой из

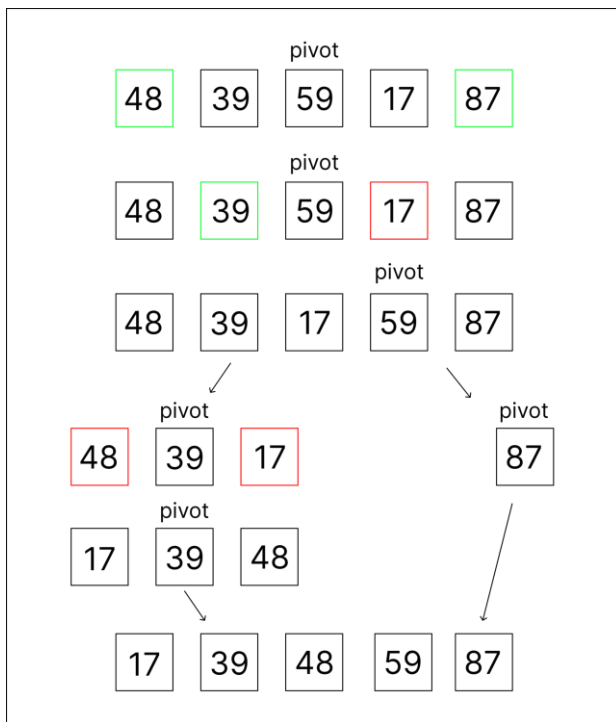
них. На каждом шаге выбирается новый опорный элемент, который помогает дальше разбивать массив на две части. Когда подмассивы достигают размера 1 или 0, сортировка завершается.

```
int partOfQuickSort(int arr[], int left, int right) {
    int opora = arr[(left + right) / 2];
    while (left <= right) {
        while (arr[left] < opora) left++;
        while (arr[right] > opora) right--;
        if (left <= right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
    }
    return left;
}

void quickSort(int arr[], int start, int end) {
    if (start >= end) return;
    int rightStart = partOfQuickSort(arr, start, end);
    quickSort(arr, start, rightStart - 1);    quickSort(arr, rightStart, end);
}
```

Эта реализация сортирует массив arr в диапазоне от left до right. Опорный элемент выбирается как средний элемент из диапазона.

Сложность быстрой сортировки в среднем случае $O(n \log n)$, в худшем случае $O(n^2)$. Однако, благодаря эффективной реализации и небольшому количеству дополнительной памяти, быстрая сортировка является одним из наиболее широко используемых алгоритмов сортировки в программировании.



Теоретические сведения № 2

List — это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а другой — на предыдущий элемент списка.

List предоставляет доступ только к началу и к концу списка — произвольный доступ запрещен. Если вы хотите найти значение где-то в середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдете то, что ищете.

Преимуществом двусвязного списка является то, что добавление элементов происходит очень быстро, если вы, конечно, знаете, куда хотите добавлять. Обычно для перебора элементов двусвязного списка используются итераторы.



Для использования данного типа списка необходимо подключить заголовочный файл `list`.

```
#include <list>
```

Чтобы объявить двусвязный список, нужно пользоваться конструкцией ниже:

```
list < тип данных > <имя контейнера>;
```

Способы создания двусвязного списка:

```
std::list<int> list1; // пустой список
std::list<int> list2(6); // список list2 состоит из 6 чисел, каждый элемент имеет
// значение по умолчанию
std::list<int> list3(4, 3); // список list3 состоит из 4 чисел, каждое число равно 3
std::list<int> list4{ 1, 2, 4, 5 }; // список list4 состоит из чисел 1, 2, 4, 5
std::list<int> list5 = { 1, 2, 3, 5 }; // список list5 состоит из чисел 1, 2, 4, 5
std::list<int> list6(list4); // список list6 - копия списка list4
std::list<int> list7 = list4; // список list7 - копия списка list4
```

В отличие от других контейнеров для типа `list` не определена операция обращения по индексу или функция `at()`, которая выполняет похожую задачу.

Тем не менее для контейнера `list` можно использовать функции `front()` и `back()`, которые возвращают соответственно первый и последний элементы.

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4, 5 };
    int first{ numbers.front() };
    int last{ numbers.back() };
    cout << "First: " << first << endl; // 1
    cout << "Last: " << last << endl; // 5
}
```

Для получения размера списка можно использовать функцию `size()`:

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
}
```

```
cout << "Size is: " << numbers.size() << endl;// 4
}
```

С помощью функции **resize()** можно изменить размер списка. Эта функция имеет две формы:

- **resize(n)**: оставляет в списке *n* первых элементов. Если список содержит больше элементов, то он усекается до первых *n* элементов. Если размер списка меньше *n*, то добавляются недостающие элементы и инициализируются значением по умолчанию

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    for (int n : numbers)// 1 2 3 4
        cout << n << "\t";
    cout << endl;
    numbers.resize(2);
    for (int n : numbers)// 1 2
        cout << n << "\t";
    cout << endl;
}
```

- **resize(n, value)**: также оставляет в списке *n* первых элементов. Если размер списка меньше *n*, то добавляются недостающие элементы со значением *value*

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    for (int n : numbers)// 1 2 3 4
        cout << n << "\t";
    cout << endl;
    numbers.resize(6, 8);
    for (int n : numbers)// 1 2 3 4 8 8
        cout << n << "\t";
}
```

```
cout << endl;
}
```

Функция **assign()** позволяет заменить все элементы списка определенным набором. Она имеет следующие формы:

- **assign(il):** заменяет содержимое контейнера элементами из списка инициализации **il**

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    numbers.assign({ 5, 6, 7, 8, 9 });
    for (int n : numbers)// 5 6 7 8 9
        cout << n << "\t";
    cout << endl;
}
```

- **assign(n, value):** заменяет содержимое контейнера **n** элементами, которые имеют значение **value**

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    numbers.assign(4, 9);
    for (int n : numbers)// 9 9 9 9
        cout << n << "\t";
    cout << endl;
}
```

- **assign(begin, end):** заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы **begin** и **end**

```
#include <iostream>
#include <list>
```

```

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    list<int> values{ 5, 6, 7, 8, 9, 10 };
    auto start = ++values.begin();
    auto end = values.end();
    numbers.assign(start, end);
    cout << "List of numbers:" << endl;
    for (int n : numbers)// 6 7 8 9 10
        cout << n << "\t";
    cout << endl;
}

```

Функция **swap()** обменивает значениями два списка:

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    list<int> values{ 5, 6, 7, 8, 9, 10 };
    numbers.swap(values);
    cout << "List of numbers:" << endl;
    for (int n : numbers)// 5 6 7 8 9 10
        cout << n << "\t";
    cout << endl;
    cout << "List of values:" << endl;
    for (int v : values)// 1 2 3 4
        cout << v << "\t";
    cout << endl;
}

```

Для добавления элементов в контейнер `list` применяется ряд функций.

- **push_back(val):** добавляет значение `val` в конец списка
- **push_front(val):** добавляет значение `val` в начало списка

```

#include <iostream>
#include <list>

```



```

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    numbers.push_back(6);
    numbers.push_front(0);
    cout << "List of numbers:" << endl;
    for (int n : numbers) // 0 1 2 3 4 6
        cout << n << "\t";
    cout << endl;
}

```

- **emplace_back(val)**: добавляет значение val в конец списка
- **emplace_front(val)**: добавляет значение val в начало списка
- **emplace(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4 };
    numbers.emplace_back(9);
    numbers.emplace_front(10);
    auto iter = ++numbers.cbegin();
    numbers.emplace(iter, 13);
    cout << "List of numbers:" << endl;
    for (int n : numbers) // 10 13 1 2 3 4 9
        cout << n << "\t";
    cout << endl;
}

```

- **insert(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент
- **insert(pos, n, val)**: вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.
- **insert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между

итераторами `begin` и `end`. Возвращает итератор на первый добавленный элемент. Если между итераторами `begin` и `end` нет элементов, то возвращается итератор `pos`.

- **`insert(pos, values)`**: вставляет список значений `values` начиная с позиции, на которую указывает итератор `pos`. Возвращает итератор на первый добавленный элемент. Если `values` не содержит элементов, то возвращается итератор `pos`.

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    std::list<int> numbers{ 1, 2, 3, 4, 5 };
    std::list<int> values{ 10, 20, 30, 40, 50 };
    auto iter1 = numbers.cbegin(); // итератор указывает на первый элемент
    numbers.insert(iter1, 0); // добавляем начало списка
    auto iter2 = numbers.cbegin(); // итератор указывает на первый элемент
    numbers.insert(++iter2, 3, 4); // добавляем после первого элемента три четверки
    auto iter3 = numbers.cbegin(); // итератор указывает на первый элемент
    numbers.insert(iter3, values.begin(), values.end()); // добавляем в начало все
    // элементы из values
    auto iter4 = numbers.cend(); // итератор указывает на позицию за последним
    // элементом
    numbers.insert(iter4, { 21, 22, 23 }); // добавляем в конец список из трех
    // элементов
    cout << "List of numbers:" << endl;
    for (int n : numbers) // 10 20 30 40 50 0 4 4 4 1 2 3 4 5 21 22 23
        cout << n << " ";
    cout << endl;
}
```

Для удаления элементов из контейнера `list` могут применяться следующие функции:

- **`clear(p)`**: удаляет все элементы
- **`pop_back()`**: удаляет последний элемент
- **`pop_front()`**: удаляет первый элемент

```
#include <iostream>
#include <list>

using namespace std;
```

```

    int main()
    {
        list<int> numbers{ 1, 2, 3, 4, 5 };
        numbers.pop_front(); // numbers = { 2, 3, 4, 5 }
        numbers.pop_back(); // numbers = { 2, 3, 4 }
        cout << "List of numbers:" << endl;
        for (int n : numbers) // 2, 3, 4
            cout << n << " ";
        cout << endl;
    }

```

- **erase(p):** удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент
- **erase(begin, end):** удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> numbers{ 1, 2, 3, 4, 5 };
    auto iter = numbers.cbegin(); // указатель на первый элемент
    numbers.erase(iter); // удаляем первый элемент
    auto begin = numbers.begin(); // указатель на первый элемент
    auto end = numbers.end(); // указатель на последний элемент
    numbers.erase(++begin, --end); // удаляем со второго элемента до
    последнего
    cout << "List of numbers:" << endl;
    for (int n : numbers) // 2 5
        cout << n << " ";
    cout << endl;
}

```

Задания к лабораторной работе № 6

Задание 1.

Реализовать возможность просмотр отсортированной структуры (использовать все рассмотренные выше сортировки). Реализовать сортировки как по убыванию, так и по возрастанию

Пример для 1 вариант. Клиент в меню помимо обычного просмотра списка номеров, может просмотреть отсортированный список номеров по их стоимости.

Задание 2.

Написать программу, в которой в начало двусвязного списка добавляется n элементов и вывести на экран.

Задание 3.

Написать программу, в которой вывести содержимое двусвязного списка в обратном порядке.

Задание 4.

Написать программу, в которой удалить все четные элементы двусвязного списка.

Контрольные вопросы

1. Что такое сортировка?
2. Причины использования сортировок.
3. Алгоритм сортировки пузырьком.
4. Алгоритм сортировки вставками.
5. Алгоритм сортировки выбором.
6. Алгоритм сортировки слиянием.
7. Алгоритм быстрой сортировки.
8. Что такое list в C++?
9. Как объявить и инициализировать list в C++?
10. Как добавить элемент в начало и конец list в C++?
11. Как удалить элемент из начала и конца list в C++?
12. Как получить доступ к элементу list по индексу в C++?
13. Как получить размер list в C++?

Дополнительные задания

Дополнительное задание 1.

Реализовать алгоритм сортировки по полю строкового типа (например, отсортировать клиентов по ФИО). Первую половину отсортировать по возрастанию, вторую по убыванию.

Дополнительное задание 2.

Реализовать алгоритм сортировки слиянием с многопоточностью (Приложение 2).

Дополнительное задание 3.

Написать программу, которая принимает на вход список (list) и удаляет из него все элементы, которые встречаются более одного раза. Результатом должен быть новый список, содержащий только уникальные элементы из исходного списка.

Лабораторная работа № 7. Методы поиска. Set

Цель: изучить способы частичного и полного удаления данных из файла; научиться редактировать массив структур, научиться работать с контейнером <set>

Теоретические сведения № 1

При работе с массивами информации очень важным является поиск информации. Существует несколько алгоритмов поиска в массиве.

Линейный поиск - это алгоритм, который мы уже неоднократно реализовывали. Он работает путем последовательного перебора всех элементов списка до тех пор, пока не будет найден искомый элемент или не будут перебраны все элементы списка.

Плюсы:

- Простота реализации.
- Работает быстро на небольших объемах данных.

Минусы:

- Неэффективен на больших объемах данных.
- Время выполнения может быть линейным в худшем случае.
- Необходимость перебирать все элементы списка в худшем случае.

Двоичный поиск - это алгоритм поиска элемента в отсортированном массиве. Он работает путем сравнения искомого элемента с элементом в середине массива. Если элемент в середине массива меньше искомого элемента, то поиск продолжается в верхней половине массива. Если элемент в середине массива больше искомого элемента, то поиск продолжается в нижней половине массива. Этот процесс повторяется до тех пор, пока искомый элемент не будет найден или пока не останется только один элемент в массиве.

Двоичный поиск имеет следующие преимущества:

- Он работает быстрее, чем линейный поиск.
- Он может быть использован для поиска элементов в больших отсортированных массивах.

Недостатки двоичного поиска:

- Массив должен быть отсортирован перед использованием двоичного поиска.
- Двоичный поиск не может быть использован для поиска элементов в неотсортированных массивах.
- Двоичный поиск может быть менее эффективным, если массив содержит много повторяющихся элементов.

Рассмотрим алгоритм двоичного поиска подробнее на примере:

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 (ключ поиска). Допустим 4. На первой итерации массив делится на две части (ищем средний элемент – midd): $(0 + 11) / 2 = 5$ (0.5 отбрасываются). Сначала, проверяется значение среднего элемента массива. Если оно совпадает с ключом – 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 шаг	///	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 шаг	///	///	///	///	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.

Пример реализации двоичного поиска:

```
#include <iostream>
using namespace std;
int binarySearch(int arr[], int n, int x) {
    int left = 0;
    int right = n - 1; // левый и правый индексы подмассива, в котором мы ищем элемент x
    while (left <= right) {
```

```

        int mid = (left + right) / 2; // вычисляем середину подмассива
        if (arr[mid] == x) { // проверяем, равен ли элемент arr[mid] элементу x
            return mid;
        }
        else if (arr[mid] < x) { //Если arr[mid] меньше, чем x, мы знаем, что x
находится в правой половине подмассива
            left = mid + 1;
        }
        else { // Если arr[mid] больше, чем x, мы знаем, что x находится в левой
половине подмассива
            right = mid - 1;
        }
    }
    return -1;
}

int main() {
    setlocale(LC_CTYPE, "rus");
    int arr[] = { 1, 3, 5, 7, 9, 11, 13, 15 };
    int n = sizeof(arr) / sizeof(arr[0]); // размер массива n
    int x = 7; // элемент x, который мы хотим найти
    int index = binarySearch(arr, n, x);
    if (index == -1) {
        cout << "Элемент не найден." << endl;
    }
    else {
        cout << "Элемент найден под индексом: " << index << endl;
    }
    // Вывод:
    // Элемент найден под индексом: 3
    return 0;
}

```

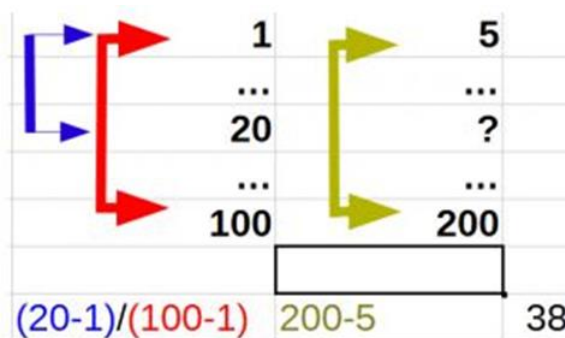

Интерполирующий поиск - это алгоритм поиска элемента в отсортированном массиве, который использует интерполяцию для нахождения близкого к искомому элемента индекса.

Он основан на принципе поиска в телефонной книге или словаре. Вместо сравнения каждого элемента с искомым, как при линейном поиске, данный алгоритм использует формулу для вычисления примерного местоположения искомого элемента в массиве. Затем он выполняет бинарный поиск вокруг этого местоположения для точного нахождения искомого элемента.

Формула расчета примерной позиции элемента в интерполяционном поиске основана на идее линейной интерполяции. Идея заключается в том, что если элементы массива равномерно распределены, то мы можем использовать линейную интерполяцию для приблизительного вычисления позиции искомого элемента в отсортированном массиве.

Интерполяция – это определение области поиска, путем вычисления подобия расстояний между искомым значением и всей областью.

Принцип работы этой формулы напоминает подобные треугольники в геометрии.



Формула достаточно проста:

1. Вычисляется длина между номерами первого элемента и искомого.
2. Такая же длина считается между первым и последним номерами.
3. Длины между собой делятся, как раз и получая вычисление подобия.
4. То же самое происходит со значениями элементов – так же вычисляется расстояние между граничными значениями в массиве.
5. Полученная длина номеров элементов массива умножается на длину значений в этих (граничащих) элементах.

6. Прибавляется значение в первой ячейке массива.

Получается: $1 + (20-1)/(100-1) * (200-5) \sim 38$

Запишем эту формулу еще раз, но теперь в общем виде:

$$\text{pos} = \text{low} + (\text{value} - \text{arr}[\text{low}]) / (\text{arr}[\text{high}] - \text{arr}[\text{low}]) * (\text{high} - \text{low})$$

где pos - позиция, на которой, возможно, находится искомый элемент, low и high - индексы первого и последнего элементов в диапазоне, value - значение искомого элемента, arr - массив данных.

Пример реализации интерполирующего поиска:

```
#include <iostream>

using namespace std;

int interpolationSearch(int arr[], int n, int x) {
    int left = 0;
    int right = n - 1; // левый и правый индексы подмассива, в котором мы
ищем элемент x
    while (left <= right && x >= arr[left] && x <= arr[right]) {
        int pos = left + ((x - arr[left]) * (right - left)) / (arr[right] - arr[left]); //
ВЫЧИСЛЯЕМ ПОЗИЦИЮ ЭЛЕМЕНТА
        if (arr[pos] == x) { // проверяем, равен ли элемент arr[mid] элементу x
            return pos;
        }
        else if (arr[pos] < x) { //Если arr[mid] меньше, чем x, мы знаем, что x
находится в правой половине подмассива
            left = pos + 1;
        }
        else { // Если arr[mid] больше, чем x, мы знаем, что x находится в левой
половине подмассива
            right = pos - 1;
        }
    }
    return -1;
}

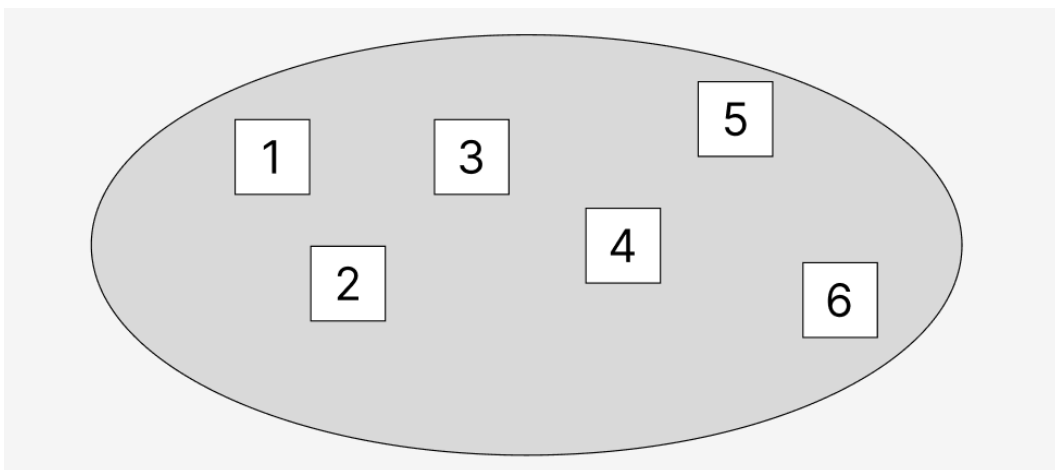
int main() {
    setlocale(LC_CTYPE, "rus");
    int arr[] = { 1, 3, 5, 7, 9, 11, 13, 15 };
    int n = sizeof(arr) / sizeof(arr[0]); // размер массива n
    int x = 7; // элемент x, который мы хотим найти
    int index = interpolationSearch(arr, n, x);
}
```

```
if (index == -1) {  
    cout << "Элемент не найден." << endl;  
}  
else {  
    cout << "Элемент найден под индексом: " << index << endl;  
}  
// Вывод:  
// Элемент найден под индексом: 3  
return 0;  
}
```

Теоретические сведения № 2

В языке C++ контейнер **set** позволяет работать с различными множествами. Под множеством в данном случае понимают некоторое количество отсортированных элементов.

Множество (set) представляет такой контейнер, который может хранить только уникальные значения. Как правило, множества применяются для создания коллекций, которые не должны иметь дубликатов. Особенностью является быстрая сортировка элементов контейнера.



Для использования данного типа списка необходимо подключить заголовочный файл `<set>`. Чтобы объявить множество, нужно пользоваться конструкцией ниже:

```
set <[тип] > <имя>;
```

Создадим множество:

```

#include <iostream>
#include <set>
using namespace std;
int main()
{
    set<int> numbers{ 2, 6, 5, 9, 0 };
    //при выводе данного контейнера получим : 0 2 5 6 9
}

```

Set является контейнером, так что имеет все стандартные для контейнера функции:

1. Работа с элементами set:
 1. swap() - меняет содержимое контейнеров местами,
 2. clear() - очистка контейнера set,
 3. count() - количество элементов в контейнере,
 4. find(a) - найти элемент a в контейнере,
 5. lower_bound(a) - первый элемент, не меньший чем a,
 6. upper_bound(a) - первый элемент, больший чем a.
2. Работа с памятью:
 - max_size() - максимальный размер контейнера.
3. Работа с контейнером:
 - begin() - указатель на начало контейнера,
 - end() - указатель на конец контейнера,
 - rbegin() - реверсивный указатель на конец контейнера,
 - rend() - реверсивный указатель на начало контейнера.

Реверсивный указатель - это вспомогательный инструмент, который может использоваться для работы с реверсивным итератором. Реверсивный итератор - это итератор, который движется в обратном направлении по контейнеру, т.е. от последнего элемента контейнера к первому элементу.

Функция **size()** возвращает количество элементов множества.

С помощью функции **empty()** можно проверить, пустое ли множество (возвращает true, если множество пусто):

```

#include <iostream>
#include <set>

```

```

using namespace std;
int main()
{
    set<int> numbers{ 2, 6, 5, 9, 0 };
    cout << "Empty: " << boolalpha << numbers.empty() << endl; //false
    cout << "Size: " << numbers.size() << endl; //5
}

```

Для добавления элементов применяется функция **insert()**:

```

#include <iostream>
#include <set>
using namespace std;
int main()
{
    set<int> numbers{ 2, 6, 5, 9, 0 };
    numbers.insert(1);
    numbers.insert(3);
    numbers.insert(5); //не добавится в множество, т.к элемент с таким
    значением уже существует
    numbers.insert(4);
    for (int n : numbers) //0 1 2 3 4 5 6 9
        cout << n << " ";
    cout << endl;
}

```

Для удаления из множества применяется функция **erase()**, в которую передается удаляемый элемент:

```

#include <iostream>
#include <set>
using namespace std;
int main()

```

```

{
    set<int> numbers{ 2, 6, 5, 9, 0 };
    numbers.erase(0);
    numbers.erase(2);
    numbers.erase(5);
    for (int n : numbers)//6 9
    cout << n << " ";
    cout << endl;
}

```

Удаление отсутствующего элемента никак не будет сказываться на работе нашей программы.

Функция **count()** позволяет проверить, есть ли определенное значение во множестве. Если определенное значение имеется во множестве, то функция возвращает 1, если нет - то 0:

```

#include <iostream>
#include <set>
using namespace std;
int main()
{
    set<int> numbers{ 2, 6, 5, 9, 0 };
    cout << "3 is in set: " << numbers.count(3) << endl; //0
    cout << "2 is in set: " << numbers.count(2) << endl; //1
}

```

Задания к лабораторной работе №7

Задание 1.

Для своей БД реализовать 2 поиска по любому полю: бинарный и интерполяционный.

Задание 2.

Написать программу, в которой нужно создать множество, добавить в него элементы и вывести их на экран в отсортированном порядке.

Задание 3.

Написать программу, в которой нужно найти элемент с минимальным значением в множестве.

Задание 4.

Написать программу, в которой нужно создать два множества, объединить их и вывести на экран общие элементы.

Контрольные вопросы

1. Какие существуют методы поиска?
2. Алгоритм двоичного поиска.
3. Алгоритм интерполяционного поиска.
4. Плюсы и минусы линейного поиска.
5. Плюсы и минусы бинарного поиска.
6. Что такое set в C++?
7. Как объявить и инициализировать set в C++?
8. Как добавить элемент в set в C++?
9. Как удалить элемент из set в C++?
10. Как проверить, содержит ли set определенный элемент в C++?
11. Как получить размер set в C++?
12. Как проверить, пустой ли set в C++?
13. Как сортировать set в C++?

Дополнительные задания

Дополнительное задание 1.

Дано действительное число a и натуральное n . Вычислите корень n -й степени из числа a , используя бинарный поиск.

Формат входных данных:

С клавиатуры через пробел вводится два числа:

1. a – действительное, неотрицательное, не превосходит 1000, задано с точностью до 6 знаков после десятичной точки;
2. n – натуральное, не превосходящее 10.

Формат выходных данных:

Требуется вывести число с точностью не менее 6 знаков после запятой.

Примеры:

Входные данные	Выходные данные
2 2	1.41421356237

Дополнительное задание 2.

Задача на поиск пересечения двух множеств. Например, пользователь вводит два множества целых чисел, программа находит их пересечение.

Дополнительное задание 3.

На основании 3 исходных множеств требуется написать функцию, которая будет возвращать симметричную разность приведенных объектов в порядке: 1-ое множество, 2-ое множество, 3-е множество.

Лабораторная работа № 8. Определение сложности алгоритма. Map

Цель: определить необходимость оценки сложности алгоритмов, изучить Big O notation как способ оценки алгоритмов, научить самостоятельно выявлять сложность, научиться работать с контейнером <map>

Теоретические сведения № 1

Оценка сложности алгоритмов - это процесс определения примерного количества ресурсов, необходимых для выполнения алгоритма, таких как время и память. Это важно, потому что некоторые алгоритмы могут потребовать значительно больше ресурсов, чем другие, что может привести к проблемам с производительностью и эффективностью.

Существует несколько способов оценки сложности алгоритмов в C++. Один из наиболее распространенных методов - это использование "большого O" нотации (Big O notation). Big O - это математическое представление о том, как быстро растет время выполнения алгоритма от размера входных данных. O-большое определяет верхнюю границу количества операций во время выполнения алгоритма для любых значений входных данных.

Для определения временной сложности алгоритма, мы анализируем количество операций, которые алгоритм выполняет в зависимости от размера входных данных. Мы учитываем только наиболее значимые операции и игнорируем константы. Затем мы представляем количество выполняемых операций в зависимости от размера входных данных в терминах асимптотического роста.

Например, если у нас есть функция, которая выполняет линейный поиск в массиве, то ее сложность может быть оценена как $O(n)$, где n - это размер массива. Это означает, что время выполнения функции будет пропорционально количеству элементов в массиве.

С другой стороны, если у нас есть функция, которая выполняет сортировку массива с использованием алгоритма быстрой сортировки, ее сложность может быть оценена как $O(n \log n)$, где n - это размер массива. Это означает, что время выполнения функции будет увеличиваться не так быстро, как размер массива, а пропорционально логарифму от размера массива.

Оценка сложности алгоритмов важна, потому что позволяет выбирать наиболее эффективный алгоритм для конкретной задачи. Она также может помочь оптимизировать код, устраняя узкие места и улучшая производительность программы.

Существуют различные типы сложности алгоритмов, которые определяются при помощи Big O:

Константная сложность

$O(1)$ - время выполнения алгоритма не зависит от размера входных данных, алгоритм выполняется за постоянное время;

```
#include <iostream>

using namespace std;

int main() {
    setlocale(LC_CTYPE, "rus");
    int x = 5, y = 10;
    int sum = x + y;
    cout << "Сумма " << x << " и " << y << " равна " << sum << endl;
    return 0;
}
```

Эта программа просто складывает два числа и выводит результат на экран. Время выполнения этой программы не зависит от количества элементов и остается постоянным, поэтому ее сложность составляет $O(1)$.

Логарифмическая сложность

$O(\log n)$ - время выполнения алгоритма растет медленнее, чем линейно, а именно, логарифмически от размера входных данных;

```
#include <iostream>

using namespace std;

int binarySearch(int arr[], int left, int right, int x) {
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] < x)
            left = mid + 1;
    }
}
```

```

        else
            right = mid - 1;
        }

        return -1;
    }

```

Алгоритм бинарного поиска работает за время, пропорциональное логарифму от размера массива n . Каждую итерацию цикла `while` мы уменьшаем диапазон поиска в два раза, пока не найдем значение x , или не поймем, что оно не присутствует в массиве. Поскольку каждая итерация цикла уменьшает диапазон поиска вдвое, мы можем найти значение x за $\log n$ итераций цикла, где n - размер массива. Таким образом, сложность алгоритма бинарного поиска составляет $O(\log n)$.

Линейная сложность

$O(n)$ - линейная сложность, время выполнения алгоритма пропорционально размеру входных данных;

```

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

int main() {
    int n, i, sum = 0;
    cout << "Введите количество элементов: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Введите элементы: ";
    for (i = 0; i < n; i++) {
        cin >> arr[i];
        sum += arr[i];
    }
    cout << "Сумма элементов " << sum << endl;
    return 0;
}

```

```
}
```

В данном примере цикл `for` с индексом `i` проходит по всем элементам массива, выполняя операции чтения из консоли и сложения. Всего цикл выполняется `n` раз, поэтому время выполнения программы пропорционально `n`, что соответствует линейной сложности $O(n)$.

Линеаризованная сложность

$O(n \log n)$ - более быстрый, чем линейный, рост времени выполнения, который характерен для многих сортировочных алгоритмов, таких как `quicksort` и `mergesort`;

```
void merge(int arr[], int left, int middle, int right) {
    int size_left = middle - left + 1;
    int size_right = right - middle;
    int* left_arr = new int[size_left];
    int* right_arr = new int[size_right];

    for (int i = 0; i < size_left; i++) {
        left_arr[i] = arr[left + i];
    }
    for (int j = 0; j < size_right; j++) {
        right_arr[j] = arr[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < size_left && j < size_right) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];
            i++;
        }
        else {
            arr[k] = right_arr[j];
            j++;
        }
    }
}
```

```

        k++;
    }

    while (i < size_left) {
        arr[k] = left_arr[i];
        i++;
        k++;
    }

    while (j < size_right) {
        arr[k] = right_arr[j];
        j++;
        k++;
    }

    delete[] left_arr;
    delete[] right_arr;
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

```

Алгоритм сортировки слиянием работает за время, пропорциональное произведению n на логарифм от n . Каждый шаг алгоритма сортировки слиянием делит массив пополам, и каждая половина сортируется рекурсивно. Затем две отсортированные половины объединяются в один отсортированный массив путем сравнения элементов каждой половины в порядке возрастания. Сравнение каждого

элемента занимает время $O(1)$, а объединение двух половин занимает время, пропорциональное их размеру. Таким образом, общее время работы алгоритма сортировки слиянием составляет $O(n \log n)$.

Квадратичная сложность

$O(n^2)$ - квадратичная сложность, время выполнения алгоритма пропорционально квадрату размера входных данных, что характерно для алгоритмов, которые используют вложенные циклы;

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Алгоритм сортировки пузырьком работает за время, пропорциональное квадрату от размера массива n . Каждый проход алгоритма сравнивает каждую пару соседних элементов и переставляет их местами, если они находятся в неправильном порядке. Каждый проход алгоритма гарантирует, что самый большой элемент массива перемещается в конец. Поскольку мы выполняем n проходов по массиву, каждый из которых требует n сравнений и, возможно, n перестановок, общее время работы алгоритма сортировки пузырьком составляет $O(n^2)$.

Экспоненциальная сложность

$O(2^n)$ - экспоненциальная сложность, время выполнения алгоритма быстро растет с увеличением размера входных данных;

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

int fib(int n) {
    if (n <= 1)
```

```
    return n;  
    return fib(n - 1) + fib(n - 2);  
}
```

Этот код реализует рекурсивную функцию для вычисления n -го числа Фибоначчи, которое определяется как сумма двух предыдущих чисел Фибоначчи.

Алгоритм вычисления чисел Фибоначчи рекурсивным методом работает за время, пропорциональное 2 в степени n . Каждый вызов функции `fib` вызывает два дополнительных вызова функции `fib`, за исключением базовых случаев, когда n меньше или равно 1 . Таким образом, количество вызовов функции `fib` экспоненциально возрастает с ростом n , и общее время работы алгоритма составляет $O(2^n)$.

Факториальная сложность

$O(n!)$ - факториальная сложность, что означает, что время выполнения алгоритма возрастает очень быстро с ростом количества входных данных.

```
#include <iostream>  
#include <algorithm>  
#include <vector>  
  
int main() {  
    int n = 4;  
    std::vector<int> v(n);  
    for (int i = 0; i < n; ++i) {  
        v[i] = i + 1;  
    }  
    do {  
        for (int i = 0; i < n; ++i) {  
            std::cout << v[i] << " ";  
        }  
        std::cout << std::endl;  
    } while (std::next_permutation(v.begin(), v.end()));  
    return 0;  
}
```

Сложность $O(n!)$ обычно возникает в алгоритмах с перебором всех возможных перестановок элементов. В этом примере мы создаем вектор из n элементов, заполненный числами от 1 до n . Затем мы вызываем функцию `std::next_permutation`, которая перебирает все возможные перестановки элементов вектора. Внутри цикла мы выводим каждую перестановку на экран. Количество перестановок для n элементов равно $n!$, что и определяет сложность алгоритма.

Получение элемента коллекции это $O(1)$. Будь то получение по индексу в массиве, или по ключу в словаре в нотации Big O это будет $O(1)$.

- Перебор коллекции это $O(n)$.
- Вложенные циклы по той же коллекции это $O(n^2)$.
- Разделяй и властвуй (Divide and Conquer) всегда $O(\log n)$.
- Итерации которые используют “Разделяй и властвуй” (Divide and Conquer) это $O(n \log n)$.

Чтобы определить сложность алгоритма в Big O, необходимо выполнить следующие шаги:

1. Определить количественный размер входных данных, на которых будет работать алгоритм. Например, для алгоритма сортировки это может быть количество элементов в массиве.

2. Определить базовую операцию, которая выполняется в алгоритме и зависит от размера входных данных. Например, для алгоритма сортировки это может быть сравнение двух элементов.

3. Оценить количество выполнений базовой операции в зависимости от размера входных данных. Например, для алгоритма сортировки пузырьком количество сравнений будет равно $n*(n-1)/2$, где n - количество элементов в массиве.

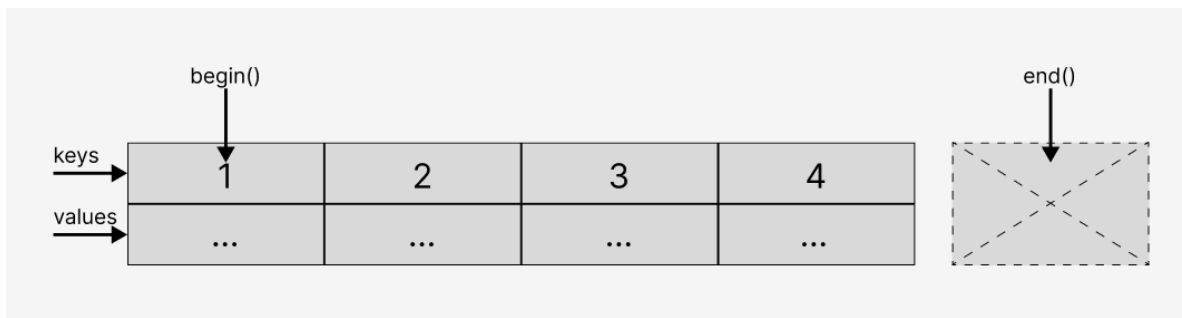
4. Определить, как зависит количество выполнений базовой операции от размера входных данных. Например, для алгоритма сортировки пузырьком сложность будет $O(n^2)$, так как количество сравнений растет пропорционально квадрату размера входных данных.

5. Если в алгоритме выполняются несколько базовых операций, необходимо определить, какие из них являются доминирующими и выбрать наибольшую сложность.

6. Проверить полученный результат для различных значений размера входных данных, чтобы убедиться, что оценка корректна.

Теоретические сведения № 2

Карта или `std::map` представляет контейнер, где каждое значение ассоциировано с определенным ключом. И по этому ключу можно получить элемент. Причем ключи могут иметь только уникальные значения.



Для использования данного типа списка необходимо подключить заголовочный файл

`<map>`.

Чтобы создать map нужно воспользоваться данной конструкцией:

```
std::map < <L>, <R> > <имя>;
```

`<L>` — этот тип данных будет относиться к значению ключа.

`<R>` — этот тип данных соответственно относится к значению.

Определим пустой словарь (каждому ключу сопоставляется его значение):

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars;
}
```

Здесь определен словарь `cars`, который будет условно хранить цену машин. Для ключей будет применяться тип `string`, а для значений - числа типа `unsigned` (условно в качестве ключа будет выступать название машины, а в качестве значения - ее цена).

Для обращения к элементам словаря - получения или изменения их значений, так же, как в массиве или векторе, применяется оператора индексирования `[]`. Только вместо целочисленных индексов можно использовать ключи любого типа в следующем виде:

```
map[ключ] = значение
```

Используем ранее созданный нами словарь.

```

#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars;
    cars["BMWx7"] = 115000;
    cars["AudiA4"] = 65000;
    cars["Bugatti"] = 2200000;

    cout << "BMW\t" << cars["BMWx7"] << endl;//115000
    cout << "Audi\t" << cars["AudiA4"] << endl;//65000
    cout << "Bugatti\t" << cars["Bugatti"] << endl;//2200000
}

```

Каждый элемент словаря фактически представляет объект типа `std::pair<const Key, Value>`, который хранит, как ключ, так и значение. В нашем случае это объект `std::pair<const std::string, unsigned int>`. И с помощью полей `first` и `second` данного объекта мы могли бы получить соответственно ключ и значение элемента

```

#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars;
    cars["BMWx7"] = 115000;
    cars["AudiA4"] = 65000;
    cars["Bugatti"] = 2200000;
    for (const auto& element : cars)
        cout << element.first << "\t" << element.second << endl;
    //AudiA4 65000
}

```

```
//BMWx7 115000
//Bugatti 2200000
}
```

Еще один способ инициализировать словарь.

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars
    {
        {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000}
    };
}
```

Для вставки элементов используется функция insert():

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars
    {
        {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000} };
    cars.insert(make_pair("Mercedes", 80000));
    cars["Mercedes123"] = 85000;
    for (const auto& element : cars)
        cout << element.first << "\t" << element.second << endl;
    //AudiA4 65000
    //BMWx7 115000
}
```

```
//Bugatti 2200000  
//Mercedes 80000  
//Mercedes123 85000  
}
```

Для удаления же элементов применяется функция **erase()**, в которую передается ключ удаляемого элемента:

```
#include <iostream>  
#include <map>  
using namespace std;  
int main()  
{  
    map<string, unsigned> cars  
    {  
        {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000}  
    };  
    cars.erase("Bugatti");  
    for (const auto& element : cars)  
        cout << element.first << "\t" << element.second << endl;  
    //AudiA4 65000  
    //BMWx7 115000  
}
```

Для получения количества элементов в словаре применяется функция **size()**. Также класс **map** имеет функцию **empty()**, которая возвращает **true**, если словарь пуст.

```
#include <iostream>  
#include <map>  
using namespace std;  
int main()  
{  
    map<string, unsigned> cars
```

```

{
    {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000}
};
cout << "Cars count: " << cars.size() << endl; //3
cout << "Cars is empty: " << boolalpha << cars.empty() << endl; //false
}

```

Чтобы проверить, есть ли в словаре элемент с определенным ключом, применяются функции **count()** (возвращает 1, если элемент есть, и 0 - если отсутствует). В функцию передается ключ элемента:

```

#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars
    {
        {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000}
    };
    cout << "BMWx7  \t" << cars.count("BMWx7") << endl; //BMWx7 1
    cout << "Mercedes\t" << cars.count("Mercedes") << endl; //Mercedes 0
}

```

Еще один способ узнать есть ли определенный элемент в словаре функция **find()**:

```

#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, unsigned> cars
    {

```

```

        {"BMWx7", 115000}, {"AudiA4", 65000}, {"Bugatti", 2200000}
    };

    map <string, unsigned> ::iterator it, it_2;
    it = cars.find("AudiA4");
    cout << it->second << endl; //65000
    it_2 = cars.find("Mersedes");
    if (it_2 == cars.end()) {
        cout << "There is no key with value 'Mersedes'";
    }
}

```

Задания к лабораторной работе № 8

Задание 1.

На каждую из вышерассмотренных сложностей написать свою программу.

Задание 2.

Сравнить скорость работы алгоритмов сортировки из лабораторной работы № 6 и определить их сложность в Big O.

Задание 3.

Написать программу, в которой нужно создать два словаря, объединить их и вывести на экран значения ключей, которые есть в обоих словарях.

Задание 4.

Написать программу, в которой нужно найти элемент с минимальным значением в словаре.

Контрольные вопросы

1. Для чего необходима оценка сложности алгоритма?
2. Что такое Big O notation?
3. Перечислите все виды сложностей алгоритмов с приведением примера.
4. Как самостоятельно определить сложность алгоритма?
5. Что такое map в C++?
6. Как объявить и инициализировать map в C++?

7. Как добавить элемент в map в C++?
8. Как удалить элемент из map в C++?
9. Как проверить, содержит ли map определенный ключ в C++?
10. Как получить размер map в C++?
11. Как проверить, пустой ли map в C++?
12. Как выполнить поиск элемент.

Дополнительные задания

Дополнительное задание 1.

Написать программу поиска минимального и максимального элементов в массиве. Сравнить скорость работы (Приложение 3) различных алгоритмов поиска, такие как перебор, деление пополам, сортировка, и определить их сложность в Big O.

Дополнительное задание 2.

Написать программу для вычисления факториала числа. Сравнить скорость работы рекурсивного и итеративного алгоритмов и определить их сложность в Big O.

Дополнительное задание 3.

Конечная цель задачи - подсчитать количество вхождений каждого слова из Q запросов в N строках. Первая строка содержит число N - количество строк. Далее следуют N строк, каждая из которых содержит одно слово. Затем следует число Q - количество запросов. Далее следуют Q строк, каждая из которых содержит одно слово - запрос. Необходимо вывести количество вхождений каждого запроса в N строках. Решать с использованием map.

Приложение 1. Изменение цвета консоли

Цвет текста в консоли можно изменять с помощью специальных функций в C++ при помощи вывода специальных управляющих символов (escape-последовательностей) в консольный поток.

Вот несколько примеров управляющих символов для изменения цвета текста в консоли:

- `\033[30m`` - черный
- `\033[31m`` - красный
- `\033[32m`` - зеленый
- `\033[33m`` - желтый
- `\033[34m`` - синий
- `\033[35m`` - фиолетовый
- `\033[36m`` - голубой
- `\033[37m`` - белый

Например, для изменения цвета текста на красный, используйте следующую команду:

```
std::cout << "\033[31mКрасный текст\n";
```

В C++ также есть стандартная библиотека `Windows.h`, которая позволяет управлять цветом текста в консоли при помощи функции `SetConsoleTextAttribute()`. Эти функции используются в Windows для изменения цвета текста и его фона при помощи указания номера цвета, например:

```
#include <iostream>
#include <Windows.h>
using namespace std;

int main() {
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN |
BACKGROUND_BLUE | BACKGROUND_RED);
    cout << "Зеленый текст на фоне синего и красного\n";
    return 0;
}
```


Приложение 2. Многопоточность

Многопоточность в C++ - это возможность создания и использования нескольких параллельно выполняющихся потоков в одном приложении. Каждый поток может выполнять свою собственную последовательность инструкций независимо от других потоков.

Многопоточность используется для увеличения производительности приложений, которые могут выполнять несколько задач одновременно. Например, веб-сервер может использовать несколько потоков для обработки запросов от нескольких клиентов одновременно.

В C++ многопоточность реализуется с помощью библиотеки потоков (Thread Library). Эта библиотека позволяет создавать, управлять и синхронизировать потоки. Класс `std::thread` является основным классом для создания и управления потоками в C++.

Вне зависимости от того, что поток будет делать и откуда он запускается, запуск потока в C++ сводится к конструированию объекта `std::thread`:

```
void do_some_work();  
std::thread my_thread(do_some_work);
```

Пример:

```
#include <iostream>  
#include <thread>  
void thread_func() {  
    // Код, который будет выполняться в потоке  
}  
int main() {  
    std::thread t(thread_func); // Создание потока  
    t.join(); // Ожидание завершения потока  
    return 0;  
}
```

Определение поведения потока в C++ позволяет контролировать его работу в процессе выполнения программы. Определить поведение можно используя один из двух методов: **`detach()`** и **`join()`**.

Используя метод `detach()`, мы сообщаем потоку, что он может выполняться самостоятельно до тех пор, пока мы не вышли из области действия основной функции. Программа не будет дожидаться завершения дополнительной функции. Как итог, поток может не успеть доделать всё, что ему было прописано.

```

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

void dopFunction() {
    for (int i = 0; i < 5; i++) {
        cout << "dopFunction итерация номер " << i + 1 << endl;
    }
}

int main() {
    setlocale(LC_CTYPE, "rus");
    thread th(dopFunction); // дополнительный поток
    th.detach(); // определение поведения потока
    for (int i = 0; i < 5; i++) {
        cout << "main итерация номер " << i + 1 << endl;
    }
}

```

В коде прописано, что сообщение должно выводиться 5 раз, как и в случае с функцией main. Однако, как мы видим, цикл в dopFunction отработал только 3 раза.

Чтобы дождаться завершения потока, следует вызвать функцию join() ассоциированного объекта std::thread. Мы можем заменить вызов th.detach() перед закрывающей скобкой тела функции вызовом th.join(), и гарантировать, что поток завершится до выхода из функции, то есть раньше уничтожения локальных переменных. Однако метод нужно вызывать в том месте кода, где мы хотим “дождаться” выполнения работы, то есть в нашем случае в конце программы.

Кроме того, при вызове join() очищается вся ассоциированная с потоком память, так что объект std::thread не связан ни с каким потоком. Это значит, что для каждого потока вызвать функцию join() можно только один раз; после первого вызова объект std::thread уже не допускает присоединения, и функция joinable() возвращает false.

```

#include <iostream>
#include <thread>

```

```

using namespace std;
void dopFunction() {
    for (int i = 0; i < 5; i++) {
        cout << "dopFunction итерация номер " << i + 1 << endl;
    }
}
int main() {
    setlocale(LC_CTYPE, "rus");
    thread th(dopFunction); // дополнительный поток
    for (int i = 0; i < 5; i++) {
        cout << "main итерация номер " << i + 1 << endl;
    }
    th.join(); // определение поведения потока
}

```

Если необходим более точный контроль над ожиданием потока, следует прибегнуть к условным переменным. Если же мы не вызовем ни одну функцию из предложенных выше, то у объекта потока будет вызван деструктор и программа попытается закрыться из другого потока, вследствие чего будет вызвана ошибка.

Приложение 3. Измерение времени выполнения кода

Время выполнения программы в C++ зависит от многих факторов, таких как алгоритм, входные данные, характеристики компьютера и др. Для измерения времени выполнения программы в C++ можно использовать различные подходы, в том числе:

1. Функция clock()

Функция clock() определяет количество единиц времени, прошедших с начала работы программы. Для измерения времени выполнения программы необходимо вызвать функцию clock() до и после выполнения алгоритма и вычислить разность между значениями. Вычисленное время обычно выражают в миллисекундах (ms). Пример:

```
clock_t start = clock();  
// выполнение алгоритма  
clock_t end = clock();  
double time = double(end - start) / CLOCKS_PER_SEC; // время в секундах
```

2. Функция std::chrono

Функции стандартной библиотеки C++ std::chrono могут использоваться для измерения времени выполнения программы с высокой точностью. Пример:

```
auto start = std::chrono::high_resolution_clock::now();  
// выполнение алгоритма  
auto end = std::chrono::high_resolution_clock::now();  
auto duration  
std::chrono::duration_cast<std::chrono::microseconds>(end - start);  
double time = static_cast<double>(duration.count()) / 1000000; // время в  
секундах
```

3. Использование таймеров

Можно использовать системные таймеры для измерения времени выполнения программы. Пример:

```
#include <sys/time.h>  
double get_time() {  
    struct timeval tv;
```

```
        gettimeofday(&tv, nullptr);  
        return double(tv.tv_sec) + double(tv.tv_usec) / 1000000;  
    }  
    double start = get_time();  
    // выполнение алгоритма  
    double end = get_time();  
    double time = end - start; // время в секундах
```

Измерение времени выполнения программы в C++ может быть полезным для оптимизации производительности и сравнения различных алгоритмов. Однако измерения могут быть ненадежными из-за влияния многих факторов, поэтому результаты следует интерпретировать с осторожностью.

Приложение 4. Стандартная библиотека C++ `algorithm`

Стандартная библиотека C++ (`<algorithm>`) - это набор полезных функций для работы с контейнерами, такими как векторы, списки, массивы и другие коллекции. Она содержит алгоритмы обработки элементов коллекции, функции для создания перестановок элементов и многие другие утилиты. Ниже приведены некоторые из основных функций библиотеки `<algorithm>`:

- `std::for_each()` - функция, которая применяет указанную функцию к каждому элементу коллекции

- `std::sort()` - функция, которая сортирует элементы коллекции

- `std::transform()` - функция, которая выполняет операцию над каждым элементом коллекции, создавая новую коллекцию с результатами

- `std::copy()` - функция, которая копирует элементы из одной коллекции в другую

- `std::accumulate()` - функция, которая вычисляет сумму элементов коллекции с помощью заданного бинарного оператора

Кроме того, стандартная библиотека C++ также включает в себя алгоритмы, такие как `std::find()`, `std::count()`, `std::max()`, `std::min()`, `std::binary_search()` и др.

Большинство функций стандартной библиотеки `<algorithm>` работают с итераторами - понятием, которое используется для перебора элементов коллекции. Это обеспечивает универсальность и применимость алгоритмов к различным типам коллекций и объектам.

Пример использования функции `std::sort()` для сортировки вектора:

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

int main() {
    vector<int> v = { 8, 1, 6, 5, 3, 2 };
    sort(v.begin(), v.end());

    for (int i : v) {
        cout << i << ' '; // Выведет: 1 2 3 5 6 8
    }
}
```

```
    }  
    return 0;  
}
```

В этом примере мы создали вектор `v` и используем функцию `std::sort()` для сортировки его элементов в порядке возрастания. Затем мы используем цикл `for` для вывода отсортированных элементов.