МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет

ЗВІТ
ПРО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 11
з навчальної дисципліни “Базові методології та технології програмування”

РЕАЛІЗАЦІЯ ПРОГРАМНИХ ЗАСОБІВ ОБРОБЛЕННЯ ДИНАМІЧНИХ СТРУКТУР ДАНИХ ТА БІНАРНИХ ФАЙЛІВ

ЗАВДАННЯ ВИДАВ:
доцент кафедри кібербезпеки
та програмного забезпечення
Доренський О. П.

ВИКОНАВ:
студент академічної групи КН-23
Клюй А.Я.

ПЕРЕВІРИВ:
викладач кафедри кібербезпеки
та програмного забезпечення
Дрєєва Г.М.

Кропивницький - 2024

МЕТА: набути ґрунтовних вмінь і практичних навичок командної (колективної) реалізації програмного забезпе- чення, розроблення функцій оброблення динамічних структур даних, використання стандартних засобів C++ для керування динамічною пам’яттю та бінарними файловими потоками.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ:

1. У складі команди ІТ-проекта розробити програмні модулі оброблення динамічної структури даних.

2. Реалізувати програмний засіб на основі розроблених командою ІТ-проекта модулів.

ВАРІАНТ № 1

ЗАВДАННЯ НА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Реалізувати електронний реєстр автомобілів регіонального сервісного центру МВС України (прізвище, ім’я, по батькові власника автомобіля, марка автомобіля, рік випуску, дата реєстрації, виданий державний номер, примітки).

За вибором працівника поліції програма забезпечує:

- виведення всього реєстру на екран або у заданий текстовий файл;
- додавання нового запису до реєстру,
- пошук запису в реєстрі за заданим державним номером (якщо запис відсутній, виводиться відповідне повідомлення);
- вилучення заданого запису з реєстру;
- завершення роботи програми з автоматичним записом реєстру у файл.

Реєстр автоматично завантажується з файлу під час запуску програми.

СКЛАД КОМАНДИ ІТ-ПРОЕКТА: Клюй Анастасія, Гребенюк Денис, Гончаренко Владислав

ХІД РОБОТИ

РОЗРОБЛЕНИЙ ПЛАН ВИКОНАННЯ ІТ-ПРОЕКТА «ЕЛЕКТРОННИЙ РЕЄСТР АВТОМОБІЛІВ»

1. **Аналіз задач ІТ-проекта та вимог до програмного забезпечення**

● Ціль проекту:

- розробка програми для автоматизації процесу ведення електроного реєстру автомобілів у регіональному центрі МВС України.

● Функціональні вимоги:

1. *Виведення* всього реєстру на екран або у заданий текстовий файл

2. *Додавання* нового запису до реєстру автомобілів з такою інформацією: прізвище, ім'я, по батькові власника, марка автомобіля, рік випуску, дата реєстрації, виданий державний номер та примітки;

3. *Пошук* запису в реєстрі за даним державним номером;

4. *Вилучення* заданого запису з реєстру.

5. *Завершення* роботи програми з автоматичним записом реєстру у файл.

6. *Автоматичне завантаження* з файлу під час запуску програми

● Нефункціональні вимоги:

1. програма повинна бути ефективною та швидкою у роботі;

2. надійність та стійкість програми до помилок та відмов.

● Дані:

- прізвище, ім'я, по батькові власника автомобіля;
- марка автомобіля, рік випуску, дата реєстрації;
- виданий державний номер;
- примітки.

Вимоги до програмного забезпечення

● Інтерфейс користувача:

- інтуїтивно зрозумілий та зручний для введення та виведення інформації.

● Продуктивність:

- швидке виконання запитів, оптимізація для великих обсягів даних.

● Сумісність:

- підтримка різних операційних систем.

● Масштабованість:

- легкість оновлення та розширення функціоналу.

● Технічні обмеження:

- врахування можливостей використовуваного обладнання та програмного середовища.

● Документація:

- повна документація коду

2. Специфікації ПЗ, концептуальні проектні рішення, архітектура програмного засобу, загальні алгоритми функціонування та інтерфейси модулів

Специфікація ПЗ

Обговорення щодо:

1) вимог до програмного забезпечення;

2) вибрання типу динамічної структури даних (список, стек, черга, дерево) для реалізації бази даних ПЗ;

3) створення заголовочного файлу;

4) розподілити підзадачі реалізації операції над динамічною структурою даних;

5) складання плану робіт проекту з урахуванням стандарту ISO/IEC 12207

Концептуальні проектні рішення

1) Модульна архітектура

• розподіл системи на незалежні модулі для спрощення розробки та тестування:

• модуль виведення реєстру - відповідає за виведення всього реєстру на екран або у заданий текстовий файл.

• модуль додавання нового запису - відповідає за додавання нового запису про автомобіль до реєстру.

• модуль пошуку за державним номером - відповідає за пошук запису в реєстрі за даним державним номером. Якщо запис не буде знайдено, модуль виведе відповідне повідомлення.

• модуль вилучення запису - відповідає за вилучення заданого запису з реєстру.

• модуль автоматичного запису реєстру у файл - відповідає за автоматичний запис у файл під час завершення роботи програми.

2) Використання списку для реалізації електронного реєстру автомобілів

• оптимізація швидкості пошуку, вставки та видалення даних

• поліпшення впорядкувння та доступної інформації

Архітектура програмного засобу

1) Модуль управління даними

● Основною структурою даних, є структура `CarRecord`

● Операції:

- виведення реєстру: функція для виведення всього реєстру на екран або у заданий текстовий файл. Ця операція дозволяє користувачу переглядати всі дані, що зберігаються у реєстрі.

- додавання нового запису: функція для додавання нового запису про автомобіль до реєстру. Ця операція дозволяє користувачу ввести дані про новий автомобіль та додати його до реєстру.

- пошук за державним номером: функція для пошуку запису в реєстрі за заданим державним номером. Ця операція дозволяє знайти інформацію про автомобіль за його номером.

- вилучення запису: функція для вилучення заданого запису з реєстру. Ця операція дозволяє користувачу вилучити інформацію про автомобіль з реєстру.

- завершення: функція для завершення роботи програми з автоматичним записом реєстру у файл. Під час завершення роботи програми всі дані про автомобілі автоматично записуються у файл, щоб забезпечити збереження інформації.

- автоматичне завантаження: під час запуску програми, автоматично завантажує реєстр автомобілів з файлу. Ця операція дозволяє користувачу продовжувати роботу з програмою без необхідності вручного завантаження даних.

2) Інтерфейс користувача

• Консольний інтерфейс: простий текстовий інтерфейс, що дозволяє користувачу вводити команди для управління записами;

• Взаємодія з користувачем: чіткі інструкції та запити для забезпечення правильного введення даних;

Загальні алгоритми функціонування

1) Виведення всього реєстру на екран або у заданий текстовий файл:

- проходження через всі записи в реєстрі;

- виведення кожного запису на екран або збереження у файл.

2) Додавання нового запису до реєстру:

- перевірка наявності даних про автомобіль;

- введення нових даних;

- збереження нового запису в реєстрі.

3) Пошук запису в реєстрі за даним державним номером:

- введення державного номера для пошуку;

- пошук в реєстрі за державним номером;

- виведення результату пошуку або відповідного повідомлення, якщо запис не знайдено.

4) Вилучення заданого запису з реєстру:

- введення даних для ідентифікації запису, який потрібно вилучити;

- пошук запису в реєстрі;

- вилучення знайденого запису.

5) Завершення роботи програми з автоматичним записом реєстру у файл:

- збереження всіх змін у реєстрі у файл.

- завершення виконання програми.

6) Автоматичне завантаження реєстру з файлу під час запуску програми:

- перевірка наявності файлу з реєстром;

- якщо файл існує, завантаження даних з файлу у пам'ять програми;

- якщо файл відсутній, створення порожнього реєстру.

3. Обгрунтування вид динамічної структури даних для реалізації бази даних ПЗ, складових її елементів (поля структури і їх типи).

Для реалізації бази даних програмного забезпечення «Електронний реєстр автомобілів» у мові програмування С++ було обрано двонапрямлений (двозв'язний) список.

Обґрунтування цього вибору наступне:

1) Швидкий доступ:

Двонапрямлені списки дозволяють швидше переміщатися як вперед, так і назад по списку. Це важливо для операцій, таких як пошук та видалення записів за державним номером, де може знадобитися звернення до попередніх записів.

2) Ефективність додавання та вилучення:

Додавання та видалення елементів у двонапрямленому списку може бути ефективними, оскільки не потрібно переміщатися через весь список для зміни посилань.

3) Простота реалізації:

Двонапрямлені списки є достатньо простими для реалізації та керуються досить зрозумілими правилами, що полегшує їх розуміння та використання.

4) Зручність використання:

Для операцій, таких як виведення всього реєстру на екран або у заданий текстовий файл, двонапрямлений список також може бути зручним, оскільки його можна ітерувати у будь-якому напрямку.

5) Навчальний аспект:

Часто використовується у курсах зі структур даних. Відповідно, він ідеально підходить для проектів, спрямованих на вивчення студентами фундаментальних принципів роботи структур даних.

Складові елементи структури

1) Прізвище власника автомобіля:

- поле структури: `lastName`

- тип даних: `string`

2) Ім'я власника автомобіля:

- поле структури: `firstName`

- тип даних: `string`

3) По батькові власника автомобіля:

- поле структури: `middleName`

- тип даних: `string`

4) Марка автомобіля:

- поле структури: `carBrand`

- тип даних: `string`

5) Рік випуску автомобіля:

- поле структури: `year`

- тип даних: `int`

6) Дата реєстрації автомобіля:

- поле структури: `registrationDate`

- тип даних: `string`

7) Державний номер автомобіля:

- поле структури: `licensePlate`

- тип даних: `string`

8) Примітки:

- поле структури: `notes`

- тип даних: `string`

4. Створення заголовочного файлу

Задача: створити заголовочний файл `struct_type_project_N.h` (N — номер варіанта завдання) з описом елементів динамічної структури даних мовою C++

5. Розподіл завдань між учасниками команди підзадачі з реалізації операцій над динамічною структурою даних відповідно до розробленої архітектури програмного засобу.

Клюй Анастасія

1) ***виведення*** всього реєстру на екран або у заданий текстовий файл: реалізація функції для виведення вмісту дерева на екран чи у текстовий файл.

2) ***додавання*** нового запису до реєстру: розробка методу, який дозволяє користувачу вводити новий запис, і додає цей запис до двійкового дерева пошуку. Це включає збір даних від користувача і створення нового вузла в дереві.

Гончаренко Владислав

1) ***пошук*** запису в реєстрі за даним державним номером: розробка функції пошуку, яка дозволяє користувачу ввести державний номер і знаходить відповідний запис у дереві, виводячи інформацію про автомобіль або повідомлення про те, що автомобіль не знайдено.

2) ***завершення*** роботи програми з автоматичним записом реєстру у файл: розробка методу, що забезпечує збереження всіх даних у файл при закритті програми та їх відновлення при наступному запуску.

Гребенюк Денис

1) ***вилучення*** заданого запису з реєстру: реалізація методу видалення запису за введеним користувачем державним номером, що вимагає впевненого видалення вузла з дерева та балансування його після видалення.

2) ***автоматичне*** завантаження реєстру з файлу під час запуску програми: розробка механізму для читання даних з файлу і відновлення структури дерева при запуску програми, забезпечуючи коректне відновлення стану реєстру.

ОТРИМАННІ АРТЕФАКТИ ПРОСЕСУ РЕАЛІЗАЦІЇ ПРОГРАМНОГО МОДУЛЯ

Main файл для ModulesKlui

- Проектування архітектури:

- визначення основних структур даних (`List`, `Node`, `CarRecord`) і функцій для роботи з ними;

- загальна структура програми, яка включає основні функції для додавання нових записів та виведення на екран або в файл.

- Детальне проектування

Специфікація функцій:

- `outputRegisterData(List *list)`: виведення всіх записів у реєстрі в файл;
- `outputRegisterData(List *list, const string &outputFileName)`: виведення всіх записів у реєстрі на консоль;
- `outputRegisterData(Node *node, basic_ostream<char> &stream)`: виведення записів з вузла списку у вказаний потік.
- `addCarInRegister(List *list)`: збір даних від користувача і додавання нового запису у реєстр.
- `addCarInRegister(List *list, CarRecord record)`: додавання нового запису у реєстр з уже зібраними даними.

- Конструювання:

- реалізація функцій, що виконують поставлені задачі;
- перевірка введення даних з використанням циклів `do-while` для забезпечення коректності введення;
- реалізація логіки для додавання нового запису в кінець двозв'язного списку.

- Тестування:

1) Проведення тестів для перевірки коректності роботи кожної функції:

- тестування введення і валідації даних;
- тестування виведення даних на консоль;
- тестування виведення даних у файл.

2) Перевірка на граничних випадках, таких як порожній список, максимальна довжина введених даних, та коректність обробки всіх полів запису.

ModulesKlui

- Проектування архітектури:

1) Визначення структур даних:

- `List`: двозв'язний список для зберігання записів автомобілів;
- `Node`: вузол списку, який містить дані про автомобіль та вказівники на наступний і попередній елементи;
- `CarRecord`: структура, яка зберігає інформацію про автомобіль.

2) Визначення інтерфейсів функцій для роботи з цими структурами:

- `outputRegisterData`: функції для виведення даних реєстру;
- `addCarInRegister`: функції для додавання нових записів.

• Специфікація функцій

Опис функцій у заголовочному файлі **`ModulesKlui.h`**:

- `void outputRegisterData(List *list)`: виведення реєстру на екран;
- `void outputRegisterData(List *list, const std::string &outputFileName)`: виведення реєстру у файл.
- `void outputRegisterData(Node *node, std::basic_ostream<char> &stream)`: Виведення даних з вузла у потік.
- `void addCarInRegister(List *list)`: збір даних від користувача і додавання нового запису.
- `void addCarInRegister(List *list, CarRecord record)`: додавання нового запису у реєстр з уже зібраними даними.

• Конструювання:

Реалізація функцій, які були специфіковані у заголовочному файлі:

- логіка для збору і валідації даних від користувача (`addCarInRegister`);
- логіка для виведення даних на консоль або у файл (`outputRegisterData`);
- логіка для додавання нових вузлів у двозв'язний список, забезпечуючи коректні зв'язки між вузлами.

• Тестування:

1) Тестування кожної функції на коректність виконання їх функціональності:

- тестування виведення даних на консоль (`outputRegisterData`);
- тестування виведення даних у файл (`outputRegisterData` з параметром **`outputFileName`**);
- тестування введення і валідації даних (`addCarInRegister`);

- тестування додавання нових записів у список (`addCarInRegister` з параметром `CarRecord`).

2) Перевірка на граничних випадках:

- порожній список;
- максимальні та мінімальні значення для полів структури `CarRecord`;
- коректність обробки різних форматів введення даних.

TestDriver

- Проектування архітектури:

1) визначення основної структури програми з меню для взаємодії з користувачем;

2) Інтеграція модулів для управління реєстром автомобілів:

- `ModulesHrebeniuk.h`
- `ModulesHoncharenko.h`
- `ModulesKlui.h`

3) Вибір формату зберігання даних у файлі `register_database`.

- Специфікація функцій:

1) функція для роботи з меню:

`void displayMenu()`: відображення меню на екран.

2) функції для управління додаванням, пошуком, видаленням та виведенням даних:

- `void addCarInRegister(List *list)`: додавання автомобіля до реєстру;
- `void searchCarInRegister(List *list, const std::string &searchLicensePlate)`: пошук автомобіля за номерним знаком;
- `void removeCarFromRegister(List *list, const std::string &removeLicensePlate)`: видалення автомобіля з реєстру;
- `void outputRegisterData(List *list)`: виведення усіх даних з реєстру на екран;
- `void saveRegisterData(List *list, const std::string &fileName)`: збереження реєстру у файл;
- `List* loadRegisterData(const std::string &fileName)`: завантаження реєстру з файлу.

3) Функція для завершення роботи програми:

`void exitApp(List *list)`: видалення динамічних даних та завершення роботи програми.

- Конструювання:

1) реалізація функцій для відображення меню і обробки вибору користувача:

- `displayMenu()`: відображає меню на екран;
- `exitApp(List *list)`: очищує пам'ять і завершує роботу програми.

2) Логіка для обробки кожного пункту меню:

- додавання нового автомобіля до реєстру;
- пошук автомобіля за номерним знаком;
- видалення автомобіля з реєстру;
- виведення всіх даних з реєстру.

3) Завантаження даних з файлу при старті програми і збереження змін при додаванні або видаленні автомобіля.

- Тестування:

1) Тестування роботи програми з різними сценаріями:

- додавання нового автомобіля та перевірка коректності збереження;
- пошук автомобіля за існуючим і неіснуючим номерним знаком;
- видалення автомобіля з реєстру та перевірка, що він більше не існує у списку;
- виведення всіх записів для перевірки коректності відображення даних

2) Перевірка на граничних випадках:

- порожній список;
- максимальні значення для номерного знаку;
- коректність обробки введених даних (наприклад, відмова при введенні надто довгого номерного знаку).

3) Тестування правильності роботи з файлами:

- збереження реєстру у файл після додавання або видалення записів;
- завантаження реєстру з файлу при запуску програми.

```
#include <fstream>
#include <iostream>
#include "ModulesKlui.h"
```

```
using namespace std;

void outputRegisterData(List *list){
    if (list->head == nullptr) {
        cout << "Реєстр порожній, додайте перший запис через меню." << endl;
        return;
    }

    outputRegisterData(list->head, cout);
}

void outputRegisterData(List *list, const string &outputFileName){
    ofstream output_file(outputFileName);
    outputRegisterData(list->head, output_file);
    output_file.close();
}

void outputRegisterData(Node *node, basic_ostream<char> &stream) {
    Node *current = node;
    while(current != nullptr) {
        stream << "
========================================================================= " << endl;
        stream << "| Прізвище: " << current->data.lastName << endl;
        stream << "| Ім'я: " << current->data.firstName << endl;
        stream << "| По батькові: " << current->data.middleName << endl;
        stream << "| Марка автомобіля: " << current->data.carBrand << endl;
        stream << "| Рік: " << current->data.year << endl;
        stream << "| Дата Реєстрації: " << current->data.registrationDate <<
endl;
        stream << "| Номерний знак: " << current->data.licensePlate << endl;
        stream << "| Примітки: " << current->data.notes << endl;
        stream << "
========================================================================= " << endl;

        current = current->next;
    }
}

void addCarInRegister(List *list) {
```

```
        CarRecord record;
        bool isValid;

        do {
            cout << "Ім'я: ";
            cin >> record.firstName;
            cin.ignore();

            isValid = !record.firstName.empty() && record.firstName.size() < 50;
            if (!isValid) {
                cout << "Помилка: ім'я не може бути порожнім та його максимальна
довжина повинна бути не більше 50 символів!" << endl;
            }
        } while (!isValid);

        do {
            cout << "Ім'я по батькові: ";
            cin >> record.middleName;
            cin.ignore();

            isValid = !record.middleName.empty() && record.middleName.size() <
50;
            if (!isValid) {
                cout << "Помилка: ім'я по батькові не може бути порожнім та його
максимальна довжина повинна бути не більше 50 символів!" << endl;
            }
        } while (!isValid);

        do {
            cout << "Прізвище: ";
            cin >> record.lastName;
            cin.ignore();

            isValid = !record.lastName.empty() && record.lastName.size() < 50;
            if (!isValid) {
                cout << "Помилка: прізвище не може бути порожнім та його
максимальна довжина повинна бути не більше 50 символів!" << endl;
            }
        } while (!isValid);
```

```
        do {
            cout << "Марка автомобіля: ";
            cin >> record.carBrand;
            cin.ignore();

            isValid = !record.carBrand.empty() && record.carBrand.size() < 50;
            if (!isValid) {
                cout << "Помилка: марка автомобіля не може бути порожньою та її
максимальна довжина повинна бути не більше 50 символів!" << endl;
            }
        } while (!isValid);

        do {
            cout << "Рік: ";
            cin >> record.year;
            cin.ignore();

            isValid = record.year >= 1885 && record.year <= 2024;
            if (!isValid) {
                cout << "Помилка: рік повинен бути числом від 1885 до 2024!" <<
endl;
            }
        } while (!isValid);

        do {
            cout << "Дата Реєстрації: ";
            cin >> record.registrationDate;
            cin.ignore();

            isValid = !record.registrationDate.empty();
            if (!isValid) {
                cout << "Помилка: дата реєстрації не може бути порожньою!" <<
endl;
            }
        } while (!isValid);

        do {
            cout << "Номерний знак: ";
            cin >> record.licensePlate;
            cin.ignore();
```

```
            isValid = !record.licensePlate.empty() && record.licensePlate.size()
< 10;
            if (!isValid) {
                cout << "Помилка: номерний знак не може бути порожнім та його
максимальна довжина повинна бути не більше 10 символів!" << endl;
            }
        } while (!isValid);

        cout << "Примітки: ";
        getline(cin, record.notes);

        addCarInRegister(list, record);
    }

    void addCarInRegister(List *list, CarRecord record){
        Node *node = new Node;
        node->next = nullptr;
        node->data = record;

        if (list->tail) {
          node->previous = list->tail;
          list->tail->next = node;
          list->tail = node;
        } else if (list->head == nullptr) {
          node->previous = nullptr;
          list->head = node;
          list->tail = node;
        }
    }
```

ВИСНОВОК

Хід роботи був спрямований на вирішення важливих завдань, що включали розробку програмних модулів для обробки динамічних структур даних та створення програмного засобу на їх основі. В процесі виконання лабораторної роботи було узгоджено робочий процес між учасниками команди, визначено функціональні вимоги до програмного продукту та розподілено завдання між

учасниками команди з урахуванням затвердженого у викладача розробленого плана виконання ІТ-проекта.

Розподіл завдань та виконанні кроки:

1) *Проектування програмних модулів:* у команді було проведено аналіз вимог до програмного продукту та розроблено архітектуру програмних модулів для ефективної роботи з динамічними структурами даних;

2) *Реалізація програмних модулів*: кожен учасник команди відповідав за реалізацію конкретного модулю, відповідаючи за функціонал модуля та його інтеграцію з іншими частинами програми;

3) *Тестування та налагодження*: після реалізації модулів було проведено тестування окремих компонентів та інтеграцію програми в цілому. Виявлені помилки та недоліки були виправлені, а програма була оптимізована для кращої продуктивності та стабільності;

4) *Інтеграція та фінальні тестування*: після успішного тестування окремих модулів їх було інтегровано у єдиний програмний засіб. Проведено фінальне тестування програми, яке підтвердило коректність роботи всієї системи та відповідність її функціональності поставленим вимогам.

Основні досягнення:

У головному файлі **TestDriver**, який використовує модулі **ModulesKlui**, реалізовані такі завдання:

1) Виведення всього реєстру на екран або у заданий текстовий файл: у функції `main` забезпечено можливість вибору пункту меню, який викликає функцію `outputRegisterData(list)`, яка виводить дані на екран. Якщо вказати файл, функція `outputRegisterData(List *list, const std::string &outputFileName)` виводить дані у текстовий файл.

2) Додавання нового запису до реєстру: пункт меню в `main`, який відповідає за додавання нового запису, викликає функцію `addCarInRegister(list)`, що дозволяє користувачу ввести дані нового запису, які потім додаються до структури даних.

У модулі **ModulesKlui** реалізовано основні функції для обробки даних реєстру:

1) Виведення всього реєстру на екран або у заданий текстовий файл:

- функція `void outputRegisterData(List *list)` викликає `outputRegisterData(node, stream)` для виведення даних на екран;

- функція `void outputRegisterData(List *list, const std::string &outputFileName)` відкриває файл для запису та викликає `outputRegisterData(node, stream)` для виведення даних у файл.

2) Додавання нового запису до реєстру:

- функція `void addCarInRegister(List *list)` збирає дані від користувача, створює новий запис і додає його до структури даних.

У **TestDriver** інтегровано всі модулі для забезпечення роботи програми:

1) Виведення всього реєстру на екран або у заданий текстовий файл: відповідна частина меню викликає функцію `outputRegisterData з ModulesKlui` для виведення даних;

2) Додавання нового запису до реєстру: відповідна частина меню викликає функцію `addCarInRegister з ModulesKlui` для додавання нових записів.

У результаті успішного виконання лабораторної роботи учасники команди набули наступні досягнення та висновки:

- **ефективна командна робота:** учасники команди успішно розробили окремі модулі для обробки динамічних структур даних, що дозволило зібрати готовий програмний продукт. Також, учасники команди продемонстрували високий рівень співпраці та взаємодопомоги, що сприяло успішному виконанню всіх завдань.

- **розробка функцій:** реалізовані функції для додавання, пошуку, видалення і виведення даних реєстру автомобілів продемонстрували вміння працювати з динамічною пам'яттю та файловими потоками;

- **поглиблення навичок програмування:** кожен учасник команди збагатив свої знання та навички у сфері розробки програмного забезпечення на мові програмування C++, зокрема, у керуванні динамічною пам'яттю та роботі з файловими потоками, а також роботи з бінарними файлами.

- **важливість тестування:** проведене тестування підтвердило коректність роботи реалізованих функцій та їх відповідність поставленим завданням. Окрім

цього, проведене тестування дало можливість переконатися, що тестування є важливим етапом розробки програмного забезпечення.

Отже, виконання даної лабораторної роботи дало змогу учасникам команди здобути цінні навички командної розробки програмного забезпечення, включаючи проектування, реалізацію та тестування модулів для обробки динамічних структур даних. Завдяки командній співпраці вдалося ефективно реалізувати всі поставлені завдання та забезпечити високу якість програмного продукту. Одержані знання та досвід будуть корисні в подальшій професійній діяльності, особливо в проектах, що вимагають спільної роботи над складними програмними системами.

ВІДПОВІДЬ НА ЗАПИТАННЯ І ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ ПІДГОТОВЛЕНОСТІ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

1. У програмуванні під часом життя динамічних змінних (або об'єктів) розуміють період часу, протягом якого об'єкт або змінна існує у пам'яті і доступна для використання у програмі. Цей час починається з моменту виділення пам'яті для об'єкта (наприклад, за допомогою оператора `new` у C++ або функції `malloc` у C) і закінчується моментом звільнення цієї пам'яті (за допомогою оператора `delete` у C++ або функції `free` у C). Приклад на мові C++:

```
 #include <iostream>

int main() {
    // Виділення пам'яті для цілочисельної змінної
    int* ptr = new int(10);

    // Використання змінної
    std::cout << "Значення: " << *ptr << std::endl;

    // Звільнення пам'яті
    delete ptr;

    // Після цього ptr вже не має вказувати на дійсну область пам'яті
    ptr = nullptr;

    return 0;
}
```

У цьому прикладі: пам'ять для змінної типу `int` виділяється за допомогою `new`. Час життя цієї динамічної змінної починається з цього моменту. Ми використовуємо змінну для виведення її значення. Пам'ять звільняється за допомогою `delete`, і час життя змінної закінчується. Для безпеки вказівник `ptr`

встановлюється в `nullptr`, що вказує на те, що він більше не вказує на жодну область пам’яті.

2. *Вказівник* - це змінна, яка зберігає адресу пам'яті іншої змінної. Вказівники можуть бути ініціалізовані на нічого (`null`), змінювати своє значення на адресу іншої змінної або маніпулювати даними за адресою, на яку вони вказують.

Коли варто й доцільно використовувати вказівники: 1) Динамічне виділення пам'яті: через функції, як `malloc, calloc` у C або `new` у C++; 2) Операції на масивах: вказівники можуть використовуватись для ітерування через масиви; 3) Передача великих структур даних у функції: для уникнення копіювання даних можна передавати покажчик на структуру; 4) Інтерфейси з використанням обробників: системні виклики часто повертають обробники ресурсів як вказівники; 5) Спільний доступ до ресурсів між різними частинами програми.

Посилання - це альтернативний спосіб доступу до змінної, яка вже була оголошена. Воно є псевдонімом для іншої змінної. Відмінність від вказівника полягає в тому, що посилання не може бути "пустим" і не змінює адресу, на яку вказує, після ініціалізації.

Коли варто й доцільно використовувати показники: 1) Модифікація аргументів функції без використання вказівників: це робить код чистішим та легшим для розуміння; 2) Перегрузка операторів: у C++ посилання дозволяють перегружати оператори, такі як присвоєння, порівняння тощо; 4) Функції, що повертають більші структури даних або класи: замість копіювання об'єкту функція може повернути посилання на існуючий об’єкт. 4) Робота з STL у C++: стандартні бібліотеки шаблонів часто використовують посилання для підвищення ефективності і зручності коду.

3. Приклад оголошення вказівника та його ініціалізації адресою статичного об’єкта:

```
#include <iostream>

int main() {
```

```
        // Оголошення змінної типу int
        int num = 10;

          // Оголошення вказівника на int та його ініціалізація адресою змінної
num
        int* ptr = &num;

        // Виведення значення вказівника та значення, на яке він посилається
        std::cout << "Значення вказівника: " << ptr << std::endl;
           std::cout << "Значення, на яке вказівник посилається: " << *ptr <<
std::endl;

        return 0;
    }
```

У цьому прикладі: оголошено змінну `num` типу `int` та проініціалізували її значенням 10. Оголошено вказівник `ptr` на ціле число `int*` та проініціалізовано його адреса змінної `num` за допомогою оператора `&`. Виведено значення вказівника та значення, на яке він посилається, за допомогою оператора `*`, який розім'є адресу, на яку посилається вказівник, і отримує значення, що зберігається за цією адресою.

4. Розмір порожнього вказівника у C або C++ залежить від архітектури системи, на якій він використовується. Зазвичай, на 32-бітних системах вказівник займає 4 байти, а на 64-бітних системах - 8 байтів. Це пов'язано з тим, що вказівник має зберігати адресу пам'яті, і ширина адреси залежить від розрядності адресації процесора. Для демонстрації, я можу написати невеликий шматок коду на C++, який виведе розмір вказівника:

```
#include <iostream>
int main() {
    int* ptr = nullptr; // Ініціалізація вказівника як порожнього
    std::cout << "Розмір вказівника: " << sizeof(ptr) << " байтів." << std::endl;
    return 0;
}
```

Тепер я можу скомпілювати і виконати цей код, щоб визначити, скільки байт пам'яті виділяється для порожнього вказівника.На моїй платформі розмір вказівника складає 8 байтів. Це відповідає 64-бітній системі, де адреси пам'яті мають 64-бітну ширину. Такий розмір дозволяє адресувати велику кількість пам'яті, що є типовим для сучасних комп'ютерних систем.

5. Вказівники в мовах C та C++ можуть бути використані для різних операцій, які відіграють ключову роль у керуванні пам'яттю та взаємодії з даними.

Ось декілька основних операцій над вказівниками і оператори, які їх реалізовують: 1) *Присвоєння*:

- казівнику можна присвоїти адресу змінної або іншого вказівника.
- Оператор: `=`
- Приклад: `int* ptr = &var;`

2) *Розіменування*:

- Отримання значення за адресою, на яку вказує вказівник.
- Оператор: `*`
- Приклад: `int value = *ptr;`

3) *Взяття адреси*:

- Отримання адреси змінної.
- Оператор: `&`
- Приклад: `int* ptr = &var;`

4) *Арифметика вказівників:*

- Зміна адреси вказівника з урахуванням типу даних, на які він вказує.
- Оператори: `+, -, ++, --`
- Приклади:

`ptr++` (збільшення адреси на розмір типу, на який вказує вказівник)

`ptr + 5` (переміщення вказівника на 5 елементів вперед)

5) *Порівняння вказівників*:

- Порівняння адрес, на які вказують вказівники.
- Оператори: `==, !=, <, >, <=, >=`
- Приклад: `if (ptr1 == ptr2`)

6) *Отримання елемента за індексом*:

- Використання вказівника як масиву для доступу до конкретного елемента.
- Оператор: `[]`
- Приклад: `int x = ptr[2];`

7) *Приведення типів*:

- Зміна типу вказівника для роботи з різними типами даних.
- Оператор: `reinterpret_cast, static_cast`
- Приклад: `char* char_ptr = reinterpret_cast<char*>(ptr);`

6. У мові програмування C++, ключові слова `new` та `delete` використовуються для управління динамічною пам'яттю. Вони дозволяють програмістам виділяти та звільняти пам'ять у кучі (heap) протягом виконання програми. Ключове слово `new` використовується для виділення пам'яті у кучі для змінних будь-якого типу. `new` не тільки виділяє пам'ять, але й автоматично викликає конструктор об'єкта (якщо це клас), ініціалізуючи виділену пам’ять. Ключове слово `delete` використовується для звільнення пам'яті, яка була раніше виділена за допомогою `new`. Воно також автоматично викликає деструктор об'єкта (якщо це клас), забезпечуючи коректне звільнення ресурсів.

7. Оператор `delete` використовується для звільнення пам'яті, яка була виділена оператором `new` для одиночного об'єкта. Коли `delete` використовується для об'єкта класу, він автоматично викликає деструктор цього об'єкта, що забезпечує належне прибирання ресурсів перед звільненням пам’яті.

Оператор `delete[]` використовується для звільнення пам'яті, яка була виділена оператором `new[]`. Цей оператор призначений для масивів об'єктів і забезпечує виклик деструктора для кожного елемента масиву перед звільненням пам'яті.

Необхідність використання `delete[]` замість `delete` випливає з того, що C++ не веде облік кількості об'єктів у масиві, тому програміст має явно вказати, що потрібно виконати очищення масиву.

8. У виразі `short ptt, *stm = &ptt;` ми маємо дві змінні: `ptt`, яка є змінною типу `short`, та `stm`, яка є вказівником на `short`. Ініціалізація `*stm = &ptt;` означає, що `stm` зберігає адресу змінної `ptt`. Вказівник `stm` вказує на пам'ять, де зберігається змінна `ptt`. Якщо ми змінимо значення `ptt` або якщо ми змінимо значення за адресою, на яку вказує `stm`, зміни будуть відображатися в обох випадках, оскільки `stm` і `ptt` відносяться до одного і того ж місця в пам’яті.

9. Ключове слово, яке використовується в мові програмування C++ для визначення розміру змінної або типу даних, це `sizeof`. Оператор `sizeof` повертає

розмір змінної або типу даних у байтах і може бути використаний як для конкретних об'єктів, так і для типів даних загалом.

10. У C та C++, унарна операція опосередкованої адресації використовує оператор розіменування `*`. Цей оператор використовується для отримання значення за адресою, на яку вказує вказівник. Основний синтаксис унарної операції опосередкованої адресації виглядає так: `*pointer;` де `pointer` – це вказівник на змінну.

11. Приклад створення динамічного одновимірного масиву в мові програмування C++ та звільнення виділеної для нього динамічної пам’яті:

```
 #include <iostream>

int main() {
    int n; // Розмір масиву
    std::cout << "Введіть розмір масиву: ";
    std::cin >> n;

    // Створення динамічного масиву
    int* array = new int[n];

    // Ініціалізація масиву
    for (int i = 0; i < n; i++) {
        array[i] = i + 1;  // Присвоюємо значення для прикладу
    }

    // Виведення елементів масиву
    std::cout << "Елементи масиву: ";
    for (int i = 0; i < n; i++) {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;

    // Звільнення динамічно виділеної пам'яті
    delete[] array;

    return 0;
}
```

У цьому прикладі:ми спочатку запитуємо у користувача розмір масиву. За допомогою оператора `new[]` виділяємо пам'ять для масиву цілих чисел розміром `n`. Ініціалізуємо масив і виводимо його елементи.Використовуємо оператор `delete[]` для звільнення виділеної пам’яті.

12. У виразі `sizeof dptr - sizeof sptr`, `sizeof dptr` поверне розмір вказівника на тип `double`, тоді як `sizeof sptr` поверне розмір вказівника на тип

short. Оскільки розмір вказівника на double буде більшим за розмір вказівника на short, то відповідь буде додатною, а конкретно це буде різниця в розмірі двох вказівників. Таким чином, в стандартний потік cout буде виведено різницю в розмірах вказівників double і short.

13. У C++, для відкриття файлового потоку в двійковому режимі використовується константа класу ios: ios::binary. Ця константа гарантує, що дані читатимуться та записуватимуться у файл без будь-якої трансформації

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

int main() {
     // Відкриття файлу для запису в двійковому режимі, додавання даних в кінець файлу
    std::ofstream file("data.bin", std::ios::binary | std::ios::app);

    // Перевірка, чи вдалося відкрити файл
    if (file.is_open()) {
        // Запис даних у файл
        int data = 12345;
        file.write(reinterpret_cast<const char*>(&data), sizeof(data));

        // Закриття файлу
        file.close();

        std::cout << "Дані успішно записано у файл." << std::endl;
    } else {
        std::cerr << "Помилка відкриття файлу." << std::endl;
    }

    return 0;
}
```

У цьому прикладі: файл "data.bin" відкривається для запису в двійковому режимі з опцією std::ios::binary і додаванням даних в кінець файлу за допомогою опції std::ios::app. Дані (у даному випадку це ціле число 12345) записуються у файл за допомогою методу write().Файл закривається після запису даних.

14. Яка функція-член об’єкта fstream C++ забезпечує читання заданої кількості байт з асоційованого потоку в змінну-буфер, а яка — включення (запис) даних у потік?

Функція-член об'єкта fstream C++, яка забезпечує читання заданої кількості байт з асоційованого потоку в змінну-буфер, називається read(). Вона

використовується для зчитування вказаної кількості байт з файлу або потоку у вказаний буфер.

Функція-член об'єкта `fstream`, яка забезпечує включення (запис) даних у потік, називається write(). Вона використовується для запису даних із буфера у файл або потік.

15. Якщо в асоційований з файлом бінарний потік буде включено рядок символів у кодуванні UTF-8, то у файл будуть записані байти, що відповідають цим символам у вигляді UTF-8 кодування**.** UTF-8 - це формат кодування символів, який використовується для представлення тексту у вигляді послідовностей байтів. У цьому форматі кожен символ може бути представлений різною кількістю байтів, зазвичай від 1 до 4 байтів, в залежності від кодової точки символу. При записі у файл, символи будуть конвертовані в їх відповідні байтові представлення у кодуванні UTF-8, і ці байти будуть записані у файл. Наприклад, рядок "Привіт, світ!" буде записаний у файл у вигляді послідовності байтів, що відповідають його UTF-8 кодуванню.

16. *Список* - це колекція елементів, організованих у лінійний порядок. Ця структура даних дозволяє динамічне додавання та видалення елементів, і може бути реалізована як зв'язний список (односпрямований або двоспрямований). Випадки використання:

- реалізація динамічних наборів елементів, де потрібен швидкий доступ до елементів на початку або в кінці.

- реалізація стеків та черг, використовуючи спеціалізовані операції.

Черга - це колекція, яка обробляє елементи за принципом FIFO (First In, First Out). Елементи додаються на один кінець черги та видаляються з іншого. Випадки використання:

- управління задачами в операційних системах, де задачі обробляються у порядку їх надходження.

- симуляція реальних черг, як от черги клієнтів або пакетів даних у мережах.

Стек - це колекція, яка працює за принципом LIFO (Last In, First Out). Елементи додаються та видаляються з одного кінця стека.

Випадки використання:

- підтримка рекурсивних викликів у програмах, зберігання адрес повернення при виклику функцій.

- алгоритми, що використовують обход в глибину (наприклад, обхід графів).

Дерево - це деревоподібна структура даних, у якій кожен вузол має не більше двох дітей. Це може бути бінарне дерево пошуку, де елементи в лівому піддереві менші за кореневий, а в правому - більші.

Випадки використання:

- побудова ефективних структур для швидкого пошуку, вставки та видалення (наприклад, бінарні дерева пошуку).

- побудова приоритетних черг та імплементація куп (heap).

17. *Однонаправлені* списки містять вузли, кожен з яких має посилання лише на наступний вузол у списку. Це означає, що ви можете переміщатися по списку тільки в одному напрямку: від голови до хвоста.

```
cpp
Copy code
struct Node {
    int data;     // Дані, що зберігаються у вузлі
    Node* next;   // Вказівник на наступний елемент списку

    Node(int value) : data(value), next(nullptr) {} // Конструктор
};

// Створення нового вузла і виділення пам'яті
Node* head = new Node(10); // створює голову списку з значенням 10
```

Двонаправлені списки містять вузли, кожен з яких має посилання на наступний і попередній вузли. Це дозволяє переміщатися по списку в обох напрямках: від голови до хвоста і навпаки. Двонаправлені списки вимагають більше пам'яті через додаткове посилання в кожному вузлі.

```
cpp
Copy code
struct DoublyNode {
    int data;           // Дані, що зберігаються у вузлі
    DoublyNode* next;   // Вказівник на наступний елемент списку
    DoublyNode* prev;   // Вказівник на попередній елемент списку

        DoublyNode(int value) : data(value), next(nullptr), prev(nullptr) {} //
Конструктор
};
```

```
// Створення нового вузла і виділення пам'яті
DoublyNode* head = new DoublyNode(10); // створює голову списку з значенням 10
```

У прикладах вище демонструється виділення динамічної пам'яті за допомогою оператора `new` для створення нових вузлів у списку. В обох випадках створюється новий вузол, ініціалізований певним значенням. Різниця полягає у додатковому вказівнику `prev` у вузлі двонаправленого списку, який дозволяє зв'язувати вузли не тільки вперед, а й назад.

18. Перелічіть допустимі операції над лінійними списками як простими динамічними структурами даних. Лінійні списки, як базові динамічні структури даних, дозволяють виконувати низку операцій, що забезпечують гнучке управління елементами списку. Ось основні операції, які можна виконувати над лінійними списками:

1) Вставка елемента:

- додавання нового елемента на початок списку.
- вставка елемента в середину списку після або перед вказаним вузлом.
- додавання елемента в кінець списку.

2) Видалення елемента:

- видалення елемента з початку списку.
- видалення елемента з середини списку.
- видалення елемента з кінця списку.

3) Пошук елемента:

- пошук елемента за значенням.
- пошук елемента за позицією.

4) Доступ до елемента:

- отримання значення елемента за певною позицією.

5) Модифікація елемента:

- зміна значення вузла.

6) Перевірка на порожнечу:

- перевірка, чи є список порожнім.

7) Отримання розміру списку:

• обчислення кількості елементів у списку.

8) Проходження по списку:

• виконання операцій над кожним елементом списку (наприклад, вивід усіх значень).

9) Очищення списку:

• видалення всіх елементів списку, що ефективно очищує список.

19) У програмуванні та обчислювальних системах термін "потік" може мати кілька значень, зокрема стосовно потоків виконання (execution threads) та потоків даних (data streams).

Потік виконання — це найменша одиниця процесу, яка може бути запланована та виконана операційною системою. Він може бути частиною процесу і має змогу виконувати код незалежно. Відмінності у використанні:

- Многозадачність: дозволяє програмі виконувати кілька завдань одночасно.

- Ефективне використання ресурсів: може підвищити продуктивність застосування шляхом паралелізації завдань, особливо на багатоядерних процесорах.

- Синхронізація: потребує уваги до синхронізації та уникнення умов змагання та взаємоблокувань.

Потік даних — це послідовність елементів даних, які доступні протягом певного часу. Це можуть бути потоки вводу/виводу, які передають дані до або з джерела або приймача (наприклад, файл, мережеве з’єднання). Відмінності у використанні:

- Послідовний доступ: дані обробляються послідовно, тобто один елемент за іншим.

- Блокування та неблокування: потоки можуть бути блокуючими або неблокуючими, залежно від методу обробки даних.

- Буферизація: часто використовується буферизація для підвищення ефективності при передачі даних.

20. Алгоритм запису в двійковий файл однонаправленого списку:

1) Відкриття файлу: відкрити двійковий файл для запису.

2) Ітерація по списку: почати з голови списку.

3) Запис даних: записати значення кожного вузла в файл.

4) Перехід: перейти до наступного вузла через вказівник next.

5) Закриття файлу: закрити файл після завершення.

Відмінності алгоритмів зберігання у файл стека, та черги:

Стек: часто записується в зворотному порядку, відповідно до принципу LIFO (Last In, First Out).

Черга: запис відбувається в порядку FIFO (First In, First Out), схоже на однонаправлений список, але може мати специфічні особливості в залежності від реалізації черги.

ВІДПОВІДЬ НА КОНТРОЛЬНІ ЗАПИТАННЯ І ЗАВДАННЯ

1. Згідно з міжнародним стандартом ISO/IEC 12207, процес комплексування (інтегрування) програмного забезпечення (ПЗ) визначається як діяльність з об'єднання різних компонентів або модулів ПЗ в єдину систему. Це означає, що окремі частини програми, які можуть бути розроблені окремо, поступово з'єднуються разом для створення цілісного програмного продукту. Процес комплексування включає в себе наступні етапи:

1) планування комплексування: визначення стратегії і плану комплексування, включаючи вибір послідовності і способів об'єднання компонентів;

2) аналіз інтеграції: оцінка вимог до інтеграції, ідентифікація залежностей між компонентами, а також аналіз ризиків і проблем, які можуть виникнути під час комплексування;

3) дизайн інтеграції: визначення способу, яким компоненти будуть об'єднуватися, розробка необхідних інтерфейсів і протоколів зв'язку між ними;

4) реалізація інтеграції: фактичне об'єднання компонентів в єдину систему, включаючи тестування цілісності системи після кожного кроку інтеграції.

5) тестування інтеграції: виконання тестів, спрямованих на перевірку правильності та надійність роботи системи після комплексування.

6) управління конфігурацією і версіонуванням: керування версіями компонентів і системи під час і після процесу комплексування;

7) завершення комплексування: підсумкове оцінювання інтеграції, а також виправлення будь-яких проблем або недоліків, що виявлені під час тестування.

2. Вказівники і посилання в мові програмування C++ є двома основними засобами, які дозволяють працювати з пам'яттю та об'єктами. Ось порівняльний аналіз обох концепцій:

Вказівники:

- синтаксис: вказівники вказують на адресу пам'яті. Наприклад, `int* ptr;` оголошує вказівник на ціле число;
- доступ до значень: для отримання значень збережених у вказівниках потрібно використовувати оператор розіменування `*`. Наприклад, `*ptr` поверне значення, на яке вказує вказівник `ptr;`
- можливість зміни адреси: вказівники можна змінювати для вказівки на інші області пам’яті;
- NULL вказівники: вказівники можуть мати значення NULL, що вказує на те, що вони не вказують на жодну область пам’яті;
- арифметика вказівників: вказівники можна інкрементувати або декрементувати, що зміщує їх вказівну адресу на кількість байт, відповідну розміру типу даних, на який вони вказують.

Посилання:

- синтаксис: посилання це "псевдонім" для об'єкта. Наприклад, `int& ref = variable;` оголошує посилання на змінну `variable;`
- доступ до значень: немає необхідності використовувати оператор розіменування. Просто використовуйте посилання як звичайну змінну;
- неможливість зміни адреси: посилання не можна перенаправити на інший об'єкт після його ініціалізації;
- NULL посилання: немає концепції "NULL посилання". Посилання завжди має вказувати на дійсний об’єкт;
- арифметика посилань: немає арифметики посилань.

3. В мові програмування C++ можна виконати ряд операцій над вказівниками. Ось деякі з найбільш поширених операцій:

1) присвоєння: вказівнику можна присвоїти адресу змінної.

```
int x = 10;
int* p = &x;
```

2) розіменування: отримання значення за адресою, на яку вказує вказівник.

```
int value = *p; // value тепер дорівнює 10
```

3) Інкрементація та декрементація: зміщення вказівника на наступну або попередню позицію в пам'яті. Ці операції беруть до уваги тип даних, на які вказує вказівник.

```
p++;   // зсув на наступну адресу типу int
p--;   // зсув на попередню адресу типу int
```

4) Арифметика вказівників: додавання або віднімання цілих чисел до або від вказівників, а також віднімання одного вказівника від іншого (що дає розмір між двома адресами в одиницях розміру об'єкта вказівника).

```
p += 1;    // перемістити p на один елемент вперед
p -= 1;    // перемістити p на один елемент назад
int* q = p + 5; // q вказує на п'ятий елемент після p
ptrdiff_t n = q - p; // n дорівнює 5
```

5) Порівняння: вказівники можна порівнювати за допомогою операторів порівняння, якщо вони вказують на елементи в межах одного і того ж масиву.

```
if (p > q) { ... }
if (p < q) { ... }
if (p == q) { ... }
```

6) Отримання адреси оператором &: мож отримати адресу змінної, щоб зберегти її в вказівнику.

```
int y = 20;
int* ptr = &y;
```

4. Операція опосередкованої адресації в мові програмування C++ використовується для отримання доступу до значення за адресою, на яку вказує вказівник. Це одна з фундаментальних можливостей мови C++, яка дозволяє пряме маніпулювання пам'яттю та забезпечує високий рівень контролю над системними ресурсами.

Синтаксис: операція опосередкованої адресації виконується за допомогою оператора розіменування `*`. Коли вказівник використовується разом з оператором `*`, це означає отримання значення за адресою, на яку вказує вказівник. Наприклад:

```
int x = 10;
int* ptr = &x;  // ptr тепер зберігає адресу змінної x
int value = *ptr;  // value дорівнює 10, *ptr розіменовує вказівник
```

У цьому прикладі:

- `int x = 10;` - звичайне оголошення змінної x з ініціалізацією значенням 10.
- `int* ptr = &x;` - `ptr` оголошується як вказівник на `int` і присвоюється йому адреса змінної x, використовуючи оператор адресації `&`.
- `int value = *ptr;` - оператор розіменування `*` використовується для отримання значення за адресою, на яку вказує ptr. Таким чином, value тепер містить те ж значення, що й x.

Основне призначення операції опосередкованої адресації в C++ полягає у забезпеченні можливості читати або модифікувати дані за певною адресою в пам'яті. Це дає змогу: 1) ефективно використовувати динамічну пам’ять; 2) створювати зв'язні структури даних, такі як списки та дерева; 3) реалізовувати алгоритми, які працюють безпосередньо з пам'яттю для оптимізації продуктивності; 4) здійснювати низькорівневе взаємодію з апаратним забезпеченням, де це потрібно.

5. У C++ клас `fstream` використовується для роботи з файлами, зокрема для їх відкриття, читання з них та запису в них. Ось основні функції-члени для кожної з цих операцій:

1) відкриття потоку: `open(const char* filename, ios_base::openmode mode)`: відкриває файл з заданим іменем і режимом.

2) запис у файл:

- `write(const char* s, streamsize n)`: записує блок байтів у файл.
- `operator<<`: записує форматовані дані в файл.

3) читання з файлу:

- `read(char* s, streamsize n)`: читає блок байтів з файлу.
- `operator>>`: читає форматовані дані з файлу.

6. При зберіганні динамічних структур у файл важливо врахувати, що зберегти можна тільки статичні дані - тобто ті, що безпосередньо містяться в полях структури. Вказівники, які часто є членами динамічних структур, не можна зберігати безпосередньо, оскільки вони містять адреси пам'яті, що будуть неактуальними після перезапуску програми або на іншій машині.

Члени структури, які підлягають зберіганню:

- примітивні типи даних: числа (цілі, дробові), булеві значення. Ці дані мають фіксований розмір і не змінюються в залежності від контексту виконання програми;

- статичні масиви з примітивними типами: ці масиви містять фіксовану кількість елементів, кожен з яких можна безпечно записати у файл.

Члени структури, які не підлягають безпосередньому зберіганню:

- вказівники: якщо структура містить вказівники на дані, ці вказівники не можна зберігати безпосередньо. Замість цього, потрібно зберігати дані, на які ці вказівники ведуть, заздалегідь обробляючи їх (наприклад, серіалізація об'єктів).

- структури чи класи, що містять динамічні дані: Це можуть бути інші об'єкти, які містять вказівники або складні структури даних. Їх також потрібно спочатку серіалізувати.

Обґрунтування: 1) вказівники мають значення тільки в контексті поточного виконання програми, оскільки вони вказують на адреси в пам'яті. Після перезапуску програми або на іншій машині ці адреси вже не будуть вказувати на ті ж області пам’яті; 2) серіалізація перетворює структури даних або об'єкти в послідовність байт, яка може бути збережена у файл і відновлена з файлу. Це дозволяє зберегти складні динамічні структури у формі, що придатна для зберігання.

7. Бінарні та текстові файлові потоки відрізняються за кількома ключовими характеристиками у способах читання та запису даних. Ось основні відмінності:

1) Форматування даних:

- текстові файли зберігають дані у формі читабельного тексту. Числа, символи та інші дані кодуються як рядки символів. Наприклад, число 123 зберігається як символи '1', '2', '3'.

- бінарні файли зберігають дані у точному бінарному форматі, в якому вони представлені в пам’яті. Число 123 може бути збережено як байтова послідовність, що відповідає його внутрішньому бінарному представленню.

2) Обробка даних:

- текстові файли часто вимагають додаткової обробки при читанні та запису, наприклад, перетворення чисел зі строкової форми в числову та навпаки. Також текстові файли мають кінцеві символи рядків, які можуть варіюватись в залежності від операційної системи (наприклад, `\n для Unix` або `\r\n` для Windows).

- бінарні файли дозволяють читати та записувати дані блоками без будь-яких перетворень, що забезпечує більшу швидкість і простоту при роботі з бінарними форматами даних, такими як зображення, аудіофайли та скомпільовані дані.

3) Ефективність:

- бінарні файли є більш ефективними з точки зору використання дискового простору і часу обробки, оскільки дані зберігаються і передаються у вигляді, який безпосередньо використовується програмами.

- текстові файли можуть бути більш зручними для користувача та легшими для налагодження і маніпуляцій у текстових редакторах, але вони зазвичай менш компактні та швидкі у використанні порівняно з бінарними файлами.

4) Універсальність:

- текстові файли легше переносити між різними платформами, але потребують врахування особливостей символів кінця рядка і кодування символів.

- бінарні файли можуть створювати проблеми сумісності при перенесенні між системами з різною архітектурою (наприклад, через відмінності у порядку байтів).

5) Використання:

- текстові файли часто використовуються для збереження конфігураційних дани, логів, документів та інших даних, які мають бути легкодоступними для читання та редагування людиною.

Бінарні файли ідеально підходять для збереженя скомпільованих програм, баз даних, мультимедійних об'єктів (наприклад, аудіо, відео, зображення), наукових даних та будь-яких інших даних, що вимагають компактності та швидкості обробки.