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”

КОМАНДНА РЕАЛІЗАЦІЯ ПРОГРАМНИХ ЗАСОБІВ ОБРОБЛЕННЯ ДИНАМІЧНИХ СТРУКТУР ДАНИХ ТА БІНАРНИХ ФАЙЛІВ

ЗАВДАННЯ ВИДАВ

доцент кафедри кібербезпеки та програмного забезпечення

Доренський О. П.

[https://github.com/odorenskyi/](https://github.com/odorenskyi/Dmytro-Parkhomenko-KB18)

ВИКОНАВ

студентка академічної групи КБ-24

Селіхова К.С.

https://github.com/Levanori

ПЕРЕВІРИВ

ст. викладач кафедри кібербезпеки   
та програмного забезпечення

Коваленко А. С.

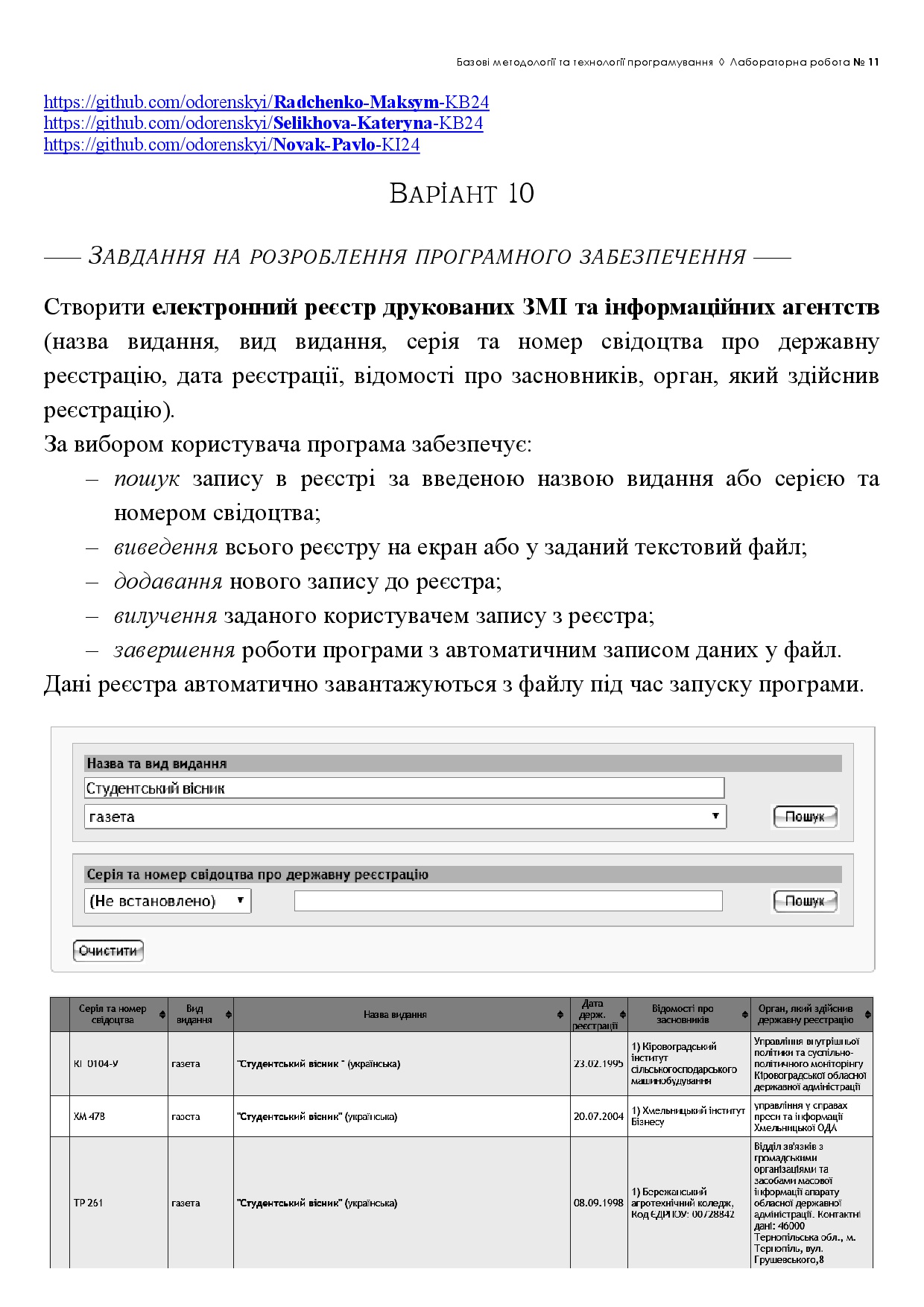
Кропивницький – 2025

**Мета роботи:** набуття ґрунтовних вмінь і практичних навичок командної (колективної) реалізації програмного забезпечення, розроблення функцій оброблення динамічних структур даних, використання стандартних засобів С++ для керування динамічною пам’яттю та бінарними файловими потоками.

**Завдання до лабораторної роботи**

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

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



**ModulesNovak:**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

struct Record {

std::string name;

std::string type;

std::string registration\_number;

std::string registration\_date;

std::string founders\_info;

std::string registering\_authority;

};

std::vector<Record> registry;

void saveRegistryData(const std::vector<Record>& registry, const std::string& filename) {

std::ofstream file(filename, std::ios::binary);

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

std::cerr << "�� ������� ������� ���� ��� ������: " << filename << std::endl;

return;

}

for (const auto& record : registry) {

file << record.name << '\n'

<< record.type << '\n'

<< record.registration\_number << '\n'

<< record.registration\_date << '\n'

<< record.founders\_info << '\n'

<< record.registering\_authority << '\n';

}

}

void loadRegistryData(std::vector<Record>& registry, const std::string& filename) {

std::ifstream file(filename, std::ios::binary);

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

std::cerr << "�� ������� ������� ���� ��� �������: " << filename << std::endl;

return;

}

Record record;

while (std::getline(file, record.name) &&

std::getline(file, record.type) &&

std::getline(file, record.registration\_number) &&

std::getline(file, record.registration\_date) &&

std::getline(file, record.founders\_info) &&

std::getline(file, record.registering\_authority)) {

registry.push\_back(record);

}

}

void outputRegistry(const std::vector<Record>& registry, const std::string& filename = "") {

std::ostream\* out;

std::ofstream file;

if (!filename.empty()) {

file.open(filename);

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

std::cerr << "�� ������� ������� ���� ��� ������: " << filename << std::endl;

return;

}

out = &file;

} else {

out = &std::cout;

}

for (const auto& record : registry) {

\*out << "����� �������: " << record.name << "\n"

<< "��� �������: " << record.type << "\n"

<< "���� �� ����� ��������: " << record.registration\_number << "\n"

<< "���� ���������: " << record.registration\_date << "\n"

<< "���������� ��� ����������: " << record.founders\_info << "\n"

<< "�����, ���� ������� ���������: " << record.registering\_authority << "\n"

<< "------------------------------" << std::endl;

}

if (file.is\_open()) {

file.close();

}

}

void exitProgram(const std::vector<Record>& registry, const std::string& filename) {

saveRegistryData(registry, filename);

std::cout << "���� ���������. �������� ���������." << std::endl;

exit(0);

}

int main() {

loadRegistryData(registry, "registry.dat");

outputRegistry(registry);

outputRegistry(registry, "exported\_registry.txt");

exitProgram(registry, "registry.dat");

return 0;

}

**ModulesNovak.h:**

#ifndef MODULENOVAK\_H\_INCLUDED

#define MODULENOVAK\_H\_INCLUDED

void saveRegistryData(const std::vector<Record>& registry, const std::string& filename);

void loadRegistryData(std::vector<Record>& registry, const std::string& filename);

void outputRegistry(const std::vector<Record>& registry, const std::string& filename = "");

void exitProgram(const std::vector<Record>& registry, const std::string& filename)

#endif // MODULENOVAK\_H\_INCLUDED

**ModulesRadchenko:**

#include <iostream>

#include <locale>

#include <string>

#include <vector>

#include <fstream>

#include <sstream>

#include <algorithm>

#include "ModulesRadchenko.h"

using namespace std;

struct MediaEntry {

string name;

string type;

string certificate;

string registration\_date;

string founders;

string registering\_authority;

};

vector<MediaEntry> registry;

// ����� ���������� �� ������

void add\_entry() {

MediaEntry entry;

cin.ignore();

cout << "����� �������: ";

getline(cin, entry.name);

cout << "��� �������: ";

getline(cin, entry.type);

cout << "���� �� ����� ��������: ";

getline(cin, entry.certificate);

cout << "���� ���������: ";

getline(cin, entry.registration\_date);

cout << "³������ ��� ����������: ";

getline(cin, entry.founders);

cout << "�����, ���� ������� ���������: ";

getline(cin, entry.registering\_authority);

registry.push\_back(entry);

cout << "����� ������ ������." << endl;

}

// ��������� ���������� �� ������

void load\_registry(const string& filename) {

ifstream file(filename);

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

cerr << "������� ��� ������� �����: " << filename << endl;

return;

}

string line;

while (getline(file, line)) {

stringstream ss(line);

MediaEntry entry;

getline(ss, entry.name, ',');

getline(ss, entry.type, ',');

getline(ss, entry.certificate, ',');

getline(ss, entry.registration\_date, ',');

getline(ss, entry.founders, ',');

getline(ss, entry.registering\_authority, ',');

if (!entry.name.empty() && !entry.type.empty()) {

registry.push\_back(entry);

} else {

cerr << "������� ����� �������� � ����: " << line << endl;

}

}

file.close();

if (file.bad()) {

cerr << "������� �� ��� ������� �����: " << filename << endl;

}

}

// ��������� ���������� � ������

void remove\_entry() {

string search\_term;

cin.ignore();

cout << "������ ����� ������� ��� ���� �� ����� �������� ��� ���������: ";

getline(cin, search\_term);

auto it = remove\_if(registry.begin(), registry.end(), [&](const MediaEntry& entry) {

return entry.name == search\_term || entry.certificate == search\_term;

});

if (it != registry.end()) {

registry.erase(it, registry.end());

cout << "����� �������� ������." << endl;

} else {

cout << "����� �� ��������." << endl;

}

}

**ModulesRadchenko.h:**

#ifndef MODULESRADCHENKO\_H\_INCLUDED

#define MODULESRADCHENKO\_H\_INCLUDED

struct MediaEntry;

void add\_entry();

void load\_registry(const string& filename);

void remove\_entry()

#endif // MODULESRADCHENKO\_H\_INCLUDED

**ModulesSelikhova:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

using namespace std;

struct MediaEntry {

string name;

string type;

string certificate;

string registration\_date;

string founders;

string registering\_authority;

};

vector<MediaEntry> registry;

void search\_by\_name() {

string search\_name;

cin.ignore();

cout << "Введіть назву видання для пошуку: ";

getline(cin, search\_name);

auto it = find\_if(registry.begin(), registry.end(), [&](const MediaEntry& entry) {

return entry.name == search\_name;

});

if (it != registry.end()) {

cout << "Запис знайдено: " << endl;

cout << "Назва: " << it->name << endl;

cout << "Тип: " << it->type << endl;

cout << "Сертифікат: " << it->certificate << endl;

cout << "Дата реєстрації: " << it->registration\_date << endl;

cout << "Засновники: " << it->founders << endl;

cout << "Орган, що видав сертифікат: " << it->registering\_authority << endl;

} else {

cout << "Запис не знайдено." << endl;

}

}

**ModulesSelikhova.h:**

#ifndef MODULESSELIKHOVA\_H\_INCLUDED

#define MODULESSELIKHOVA\_H\_INCLUDED

void search\_by\_name()

#endif // MODULESSELIKHOVA\_H\_INCLUDED

**Лістинінг застосунку prj-7-Selikhova:**

#include <iostream>

#include "Modules[Novak](https://github.com/odorenskyi/Novak-Pavlo-KI24).h"

#include "ModulesRadchenko.h"

#include "ModulesSelikhova.h"

int main() {

setlocale(LC\_ALL, "");

load\_registry\_data(registry, "registry.dat");

int choice;

while (true) {

cout << "\nМеню:\n";

cout << "1. Додати запис\n";

cout << "2. Видалити запис\n";

cout << "3. Шукати запис за назвою\n";

cout << "4. Завантажити реєстр з файлу\n";

cout << "5. Вивести реєстр\n";

cout << "6. Зберегти реєстр\n";

cout << "7. Вийти з програми\n";

cout << "Виберіть опцію: ";

cin >> choice;

switch (choice) {

case 1:

add\_entry();

break;

case 2:

remove\_entry();

break;

case 3:

search\_by\_name();

break;

case 4:

loadRegistryData("registry.txt");

break;

case 5:

outputRegistry(registry);

break;

case 6:

save\_registry\_data(registry);

break;

case 7:

exitProgram(registry, "registry.dat");

break;

default:

cout << "Невірний вибір. Спробуйте ще раз." << endl;

}

}

return 0;

}

**Результати тестування:**

1. *Для* search\_by\_name

(Позитивні тести)

* Пошук існуючого запису
* Пошук запису з особливими символами

(Негативні тести)

* Пошук неіснуючого запису
* Пошук з порожнім веденням

1. *Для* save\_registry\_data

(Позитивні тести)

* Збереження реєстру з кількома записами
* Збереження порожнього реєстру

(Негативні тести)

* Збереження з недоступною назвою файлу
* Збереження реєстру з некоректними даними

Аргументи:

1. Засвоєння основ модульного програмування: Успішне створення окремих функцій і їх об’єднання в єдину систему свідчать про розуміння модульної структури програм.
2. Формування вмінь інтеграції програмних компонентів: Об’єднання функцій у завершений консольний додаток демонструє навички побудови злагодженого програмного продукту.
3. Опрацювання навичок роботи з Git: Регулярне використання Git-репозиторію для керування кодом підтверджує володіння системами контролю версій.
4. Створення погоджених інтерфейсів модулів: Розроблення та дотримання чітких інтерфейсів у заголовкових файлах є невіддільною частиною командної роботи над програмою.
5. Участь в архітектурному плануванні: Обговорення та погодження архітектури на командних зустрічах впливає на логіку та структуру всієї програми.
6. Раціональний розподіл завдань у команді: Розподілення підзадач із реалізації динамічної структури даних свідчить про розуміння принципів декомпозиції.
7. Гнучкість у зміні власного коду: Вміння адаптувати написаний код до рішень і стандартів команди вказує на командну зрілість.
8. Опанування Code::Blocks та аналізу чужого коду: Перегляд і аналіз коду інших учасників команди сприяють вдосконаленню власного стилю програмування.
9. Дотримання стилю оформлення коду: Уніфіковане оформлення коду в рамках команди покращує читаність і підтримку програмного продукту.
10. Практика розв’язання конфліктів при злитті: Зіткнення з merge-конфліктами й успішне їх вирішення формують важливі навички командної розробки.
11. Створення статичних бібліотек у C++: Інкапсуляція функцій у статичні бібліотеки демонструє вміння створювати багаторазово використовувані компоненти.
12. Робота із заголовковими файлами: Оголошення структур і функцій у файлах .h є базовим елементом для модульної побудови коду на C++.
13. Застосування стандарту ISO/IEC 12207: Реалізація проєкту відповідно до міжнародного стандарту забезпечує системність і документованість.
14. Підготовка програмного коду до тестування: Розроблення тестів і адаптація функцій під них формують культуру перевірки працездатності.
15. Особиста відповідальність за якість коду: Кожен розробник відповідає за коректну і стабільну роботу свого внеску в проєкт.
16. Опанування механізмів динамічного виділення пам’яті: Коректне використання new та delete свідчить про розуміння керування ресурсами.
17. Програмування операцій над зв’язаними списками: Реалізація базових функцій для списку демонструє впевнене володіння покажчиками.
18. Забезпечення правильного очищення пам’яті: Реалізація функцій на кшталт clearList() допомагає уникнути витоків пам’яті.
19. Реалізація ефективного пошуку: Створення функції search\_by\_state\_number() демонструє навички побудови швидких алгоритмів пошуку.
20. Розробка функцій видалення елементів: Коректне оновлення покажчиків при видаленні елементів гарантує цілісність структури.
21. Глибоке розуміння роботи з покажчиками: Складні дії з динамічними структурами свідчать про впевнене володіння покажчиками.
22. Обробка особливих ситуацій у коді: Врахування ситуацій, як-от порожній список або крайні елементи, підвищує надійність програми.
23. Розробка абстрактних типів даних: Визначення структур у заголовкових файлах вказує на засвоєння принципів інкапсуляції.
24. Передача динамічних структур у функції: Вірне використання покажчиків чи посилань для передавання структур забезпечує ефективну взаємодію.
25. Оптимізація продуктивності коду: Обґрунтований вибір динамічної структури сприяє пришвидшенню ключових операцій програми.
26. Вміння налагоджувати помилки пам’яті: Успішне вирішення проблем з пам’яттю (наприклад, segmentation fault) підтверджує навички налагодження.
27. Реалізація повного обходу структури: Написання функцій для перегляду чи збереження всіх елементів — приклад практичної реалізації алгоритмів обходу.
28. Застосування структур і класів: Побудова елементів за допомогою структур або класів демонструє використання об’єктно-орієнтованих підходів.
29. Розуміння будови “вузла”: Успішна реалізація вузлів зі зв’язками між ними свідчить про правильне оперування базовими елементами списку.
30. Уникнення “висячих” покажчиків: Запобігання зверненню до вже звільненої пам’яті гарантує стабільність роботи програми.
31. Глибоке розуміння зв’язку пам’яті з покажчиками: Робота з покажчиками допомагає усвідомити, як саме зберігаються і обробляються дані.
32. Раціональне використання пам’яті: Оптимізація пам’яті дозволяє уникати фрагментації та покращує продуктивність програми.
33. Розробка складних алгоритмів (наприклад, сортування): Реалізація складних функцій доводить високий рівень володіння алгоритмічним мисленням.
34. Гнучкість у розвитку програми: Код, побудований на динамічних структурах, легко розширюється новими можливостями.
35. Розуміння абстрагування даних: Чітке розмежування між інтерфейсом і реалізацією дозволяє підвищити модульність.
36. Коректне використання масивів у динамічній пам’яті: Застосування new[] і delete[] вказує на вміння працювати з масивами у динаміці.
37. Запобігання витокам пам’яті: Досягнення того, що програма не спричиняє втрат пам’яті, є ознакою правильної реалізації.
38. Перевірка успішного виділення пам’яті: Контроль за результатом new запобігає розіменуванню некоректних покажчиків.
39. Усвідомлення життєвого циклу об’єктів у динаміці: Розуміння, коли створюються та знищуються об’єкти в пам’яті, допомагає точніше керувати ресурсами.
40. Обробка помилок при нестачі пам’яті: Застосування try-catch для перехоплення std::bad\_alloc підвищує надійність програми.
41. Свідоме використання покажчиків і посилань: Розуміння, коли доцільно застосовувати покажчики або посилання, дозволяє писати ефективнішу й безпечнішу програму.
42. Усвідомлене використання покажчиків та посилань: Вміння обирати між покажчиками та посиланнями відповідно до контексту функції чи алгоритму свідчить про глибоке розуміння семантики C++.
43. Розробка зрозумілого та підтримуваного коду: Написання коду з дотриманням принципів читабельності та структурованості полегшує його майбутнє супроводження та розширення.
44. Використання умовної компіляції: Застосування директив препроцесора (#ifdef, #ifndef, #define, #endif) для уникнення подвійного включення заголовків або адаптації коду до різних умов виконання.
45. Коментування коду відповідно до стандартів: Вміння надавати змістовні коментарі до складних ділянок коду підвищує його зрозумілість для інших членів команди.
46. Формування звички до рефакторингу: Регулярне вдосконалення вже написаного коду (без зміни функціональності) демонструє відповідальний підхід до якості програмного забезпечення.
47. Розмежування функціональності за допомогою модулів: Винесення окремих частин логіки в незалежні модулі забезпечує повторне використання коду та спрощує тестування.
48. Дотримання принципу "одна функція — одна задача": Розроблення функцій з чітко визначеною метою підвищує зрозумілість та зменшує ризик помилок.
49. Забезпечення узгодженості назв: Використання єдиних підходів до іменування змінних, функцій та структур полегшує навігацію у великому коді.
50. Адаптація до командних вимог документації: Ведення та підтримка документації до коду (README, Doxygen-коментарі) за домовленістю з командою підвищує прозорість проекту.
51. Розуміння принципів інкапсуляції: Створення структур або класів із чітким поділом між публічними й приватними членами демонструє знання основ об’єктно-орієнтованого програмування.
52. Організація проєкту за допомогою багатофайлової структури: Використання заголовкових (.h) та реалізаційних (.cpp) файлів для розділення оголошень і реалізацій підвищує масштабованість та зручність супроводу проєкту.
53. Створення та використання користувацьких типів: Визначення власних структур і типів даних (через struct, class, typedef, using) дозволяє точніше моделювати об'єкти прикладної області.
54. Обробка нестандартних ситуацій: Реалізація обробки помилок (через перевірки або винятки) дозволяє програмі надійно працювати навіть у непередбачуваних умовах.
55. Використання стандартної бібліотеки (STL): Застосування контейнерів (vector, map, set тощо) і алгоритмів (sort, find, count) свідчить про володіння сучасними можливостями мови.
56. Побудова програм з використанням рекурсії: Реалізація рекурсивних алгоритмів (наприклад, обходу дерева чи обчислення факторіалу) демонструє гнучкість мислення та розуміння стекової моделі виконання.
57. Ефективне використання динамічної пам’яті: Застосування new і delete (або smart pointers у сучасному C++) для створення об'єктів під час виконання програми показує володіння механізмами керування пам’яттю.
58. Реалізація абстрактних типів даних (ADT): Побудова власних структур на кшталт списків, стеків, черг і дерев дає змогу краще зрозуміти принципи організації даних і алгоритмів над ними.
59. Застосування перевантаження функцій і операторів: Використання можливостей C++ для створення декількох версій функцій з однаковими іменами, або перевантаження операторів (+, [], ()) для роботи з власними класами.
60. Побудова інтерфейсів через чисті віртуальні функції: Створення абстрактних класів із чистими віртуальними методами (= 0) дозволяє реалізовувати поліформізм — одну з основ об'єктно-орієнтованого підходу.
61. Формування звички до тестування функцій: Написання простих тестів для перевірки коректності роботи функцій і модулів допомагає виявляти помилки на ранніх етапах.
62. Розуміння областей видимості: Знання про локальні, глобальні та статичні змінні допомагає правильно організовувати дані та уникати конфліктів і помилок.
63. Розуміння життєвого циклу змінних: Усвідомлення, коли створюються й знищуються автоматичні, статичні й динамічні змінні, дозволяє ефективно керувати пам’яттю.
64. Організація циклів за допомогою умов і лічильників: Гнучке використання for, while і do-while для побудови циклічних процесів у програмах.
65. Побудова умовних розгалужень: Ефективне використання if, else, else if, switch для управління виконанням програми на основі різних умов.
66. Оптимізація коду: Вміння зменшувати обчислювальну складність (часову та просторову) шляхом усунення надлишкових обчислень і використання ефективних структур даних.
67. Використання констант: Оголошення незмінних значень через const або constexpr допомагає захистити дані від випадкової зміни та поліпшує читабельність коду.
68. Визначення макросів і директив: Застосування директив препроцесора (#define, #ifdef, #pragma once) для керування компіляцією коду.
69. Підключення зовнішніх бібліотек: Навички додавання сторонніх бібліотек (наприклад, для роботи з графікою, мережею чи базами даних) розширюють можливості програми.
70. Використання іменованих просторів (namespace): Групування імен у простори імен (namespace) допомагає уникати конфліктів між іменами змінних, функцій і класів.
71. Виведення й форматування даних: Використання iomanip, setw, setprecision тощо для керування виглядом виведених даних.
72. Взаємодія з користувачем: Створення зрозумілого інтерфейсу введення/виведення — важливий аспект користувацького досвіду при розробці програм.
73. Реалізація простих графічних інтерфейсів: Застосування бібліотек (наприклад, SFML або Qt) для побудови елементів GUI у C++.
74. Обробка винятків: Застосування конструкцій try, catch, throw для обробки помилок і небажаних ситуацій у програмі.
75. Модульність коду: Розподіл програми на незалежні модулі та файли сприяє повторному використанню коду й полегшує його тестування.