Міністерство освіти і науки України

Відокремлений структурний підрозділ

«Зборівський фаховий коледж

Тернопільського національного технічного університету імені Івана Пулюя»

Циклова комісія комп’ютерно

інформаційних технологій

Курсова робота

з дисципліни

«Об’єктно-орієнтоване програмування» на тему:

«Дослідити можливості

вказівників в мові С++»

Розробив студент

групи 31-ІПЗ

Ігор Федорняк

Керівник проєкту

Баран М. П.

Зборів, 2025 р.

# ****ЗМІСТ****

**Завдання на курсову роботу** .................................................................................. 2

**Перелік умовних позначень** ................................................................................. 4

**Вступ** ....................................................................................................................... 5 1 **Аналіз предметної області, методів та засобів** ................................................ 7

1.1 Концептуальні основи вказівників та управління пам'яттю в C++ ............ 7

1.2 Вказівники та динамічні аспекти C++ ........................................................... 8

1.3 Перетворення та приведення типів вказівників ........................................... 9

1.4 Проблеми безпеки та сучасні практики роботи з вказівниками ............... 11

1.5 Методологія дослідження можливостей вказівників ................................. 13

1.6 Інструментальні засоби для аналізу та роботи з вказівниками ................. 13

2 **Реалізація програмного забезпечення** ............................................................ 15

2.1. **Постановка задачі** ......................................................................................... 15

2.1.1 Завдання до КР ............................................................................................ 18

2.1.2 Вхідна інформація ...................................................................................... 21

2.1.3 Вихідна інформація .................................................................................... 24  
2.2. **Опис мови програмування** ........................................................................... 28  
2.3. **Опис алгоритму та ідентифікатори** ............................................................ 33  
2.4. **Тестування програми** ................................................................................... 38

**Висновки** .............................................................................................................. 40

**Перелік використаної літератури** ....................................................................... 41

**Додатки** ................................................................................................................. 42

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ**

RAII (Resource Acquisition Is Initialization) — принцип управління ресурсами в C++, при якому ресурс захоплюється в конструкторі та звільняється в деструкторі.

STL (Standard Template Library) — стандартна бібліотека шаблонів у C++, яка містить контейнери, алгоритми та ітератори.

HFT (High-Frequency Trading) — високочастотна торгівля, приклад сфери застосування C++.

IDE (Integrated Development Environment) — інтегроване середовище розробки (наприклад, Visual Studio, CLion).

GDB (GNU Debugger) — зневаджувач для програм на C/C++ у середовищі Linux.

LLDB — альтернативний зневаджувач, зазвичай використовується в macOS.

ASan (AddressSanitizer) — інструмент для виявлення помилок пам’яті.

MSan (MemorySanitizer) — виявляє використання неініціалізованої пам’яті.

TSan (ThreadSanitizer) — виявляє умови гонки в багатопотокових програмах.

CMake — система автоматизованої збірки програм.

Make — класичний інструмент для збірки програм у Unix-подібних системах.

VS Code (Visual Studio Code) — редактор коду з підтримкою C++.

nullptr — константа для позначення нульового вказівника (введена в C++11).

C++11, C++14, C++17, C++20, C++23 — версії стандарту мови C++.

**ВСТУП**

Мова програмування C++ надає розробникам потужні інструменти для створення високоефективних та гнучких програмних систем. Одним із таких ключових інструментів, що значною мірою визначає специфіку та можливості мови, є механізм вказівників. Вказівники це по суті змінні, що зберігають адреси інших об'єктів у пам'яті комп'ютера, відкривають прямий доступ до управління пам'яттю, що є критично важливим для системного програмування, розробки ігрових рушіїв, оптимізації продуктивності та реалізації складних алгоритмів і структур даних, таких як зв'язні списки, дерева та графи.

Актуальність дослідження можливостей вказівників у C++ зумовлена їхньою фундаментальною роллю в мові та широким спектром завдань, які вони дозволяють вирішувати. Вони є основою для реалізації динамічного виділення пам'яті, побудови поліморфних систем за допомогою віртуальних функцій, взаємодії з апаратним забезпеченням та С-бібліотеками, а також для ефективної роботи з масивами та рядками. Без глибокого розуміння вказівників неможливо повною мірою опанувати мову C++ та ефективно використовувати її потенціал.

Водночас, потужність вказівників нерозривно пов'язана зі значними ризиками. Неправильне їх використання — розіменування нульових або неініціалізованих вказівників, вихід за межі виділеної пам'яті, витоки пам'яті через некоректне звільнення ресурсів, використання висячих вказівників — є поширеним джерелом важковідтворюваних помилок, що можуть призводити до збоїв програми, пошкодження даних та вразливостей безпеки. Це вимагає від програміста дисципліни, уважності та чіткого розуміння принципів роботи з пам'яттю.

Метою даної курсової роботи є дослідження основних можливостей, які надають вказівники в мові C++. У роботі будуть розглянуті фундаментальні концепції, такі як оголошення вказівників, операції взяття адреси та розіменування, арифметика вказівників та їхній тісний зв'язок із масивами. Особливу увагу буде приділено механізмам динамічного виділення та звільнення пам'яті за допомогою операторів new та delete. Також будуть розглянуті такі аспекти, як вказівники на функції, використання const з вказівниками, поняття void\* та nullptr. На додаток, буде коротко згадано про сучасні підходи до безпечного управління ресурсами за допомогою розумних вказівників як альтернативи "сирим" вказівникам у багатьох сценаріях.

Дана робота покликана систематизувати знання про вказівники в C++, продемонструвати їхню універсальність та потужність на практичних прикладах, а також окреслити ключові аспекти їх безпечного та ефективного застосування в сучасній розробці програмного забезпечення.

**1** **АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ,**

**МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАВДАННЯ**

Даний розділ курсової роботи присвячений комплексному аналізу предметної області, пов'язаної з вказівниками в мові програмування C++, а також визначенню методології та інструментарію, що будуть застосовані для досягнення мети роботи — дослідження можливостей вказівників. Розуміння фундаментальних концепцій, пов'язаних з організацією пам'яті та принципами роботи вказівників, є необхідною передумовою для їх ефективного та безпечного використання. Аналіз методів дослідження дозволить обрати найбільш адекватні підходи для вивчення їхніх властивостей та сценаріїв застосування. Огляд доступних засобів допоможе визначити технічну базу для практичної частини дослідження.

**1.1 Концептуальні основи вказівників та управління пам'яттю в C++**

Фундаментальне розуміння вказівників починається з усвідомлення того, як C++ організовує та адресує пам'ять. Модель пам'яті C++, успадкована багато в чому від C, надає програмісту доступ до різних областей зберігання даних, кожна з яких має свої характеристики та призначення. Основні сегменти включають стекову пам'ять (stack), що використовується для автоматичного зберігання локальних змінних та параметрів функцій з обмеженим часом життя; динамічну пам'ять (heap або free store), призначену для об'єктів, час життя яких контролюється програмістом вручну або за допомогою спеціальних механізмів; та статичну/глобальну пам'ять для даних, що існують протягом усього часу виконання програми. Кожен байт у цих областях пам'яті має унікальну числову адресу[5].

Саме в цьому контексті з'являється вказівник: це змінна, основним призначенням якої є зберігання адреси іншої змінної або об'єкта в пам'яті. Таким чином, вказівники реалізують механізм непрямої адресації (indirection), дозволяючи програмі працювати з даними не безпосередньо, а через їхнє місцезнаходження. Це надає значну гнучкість, уможливлюючи, наприклад, зміну зовнішніх змінних з функції або створення складних динамічних структур.

Оголошення вказівника (наприклад, int\* ptr;) не лише резервує пам'ять для самої змінної-вказівника, але й вказує компілятору на тип даних, що очікуються за адресою, яку цей вказівник буде зберігати. Ця типізація є важливою для коректної інтерпретації бітів пам'яті та для виконання масштабованої арифметики вказівників. Особливим є тип void\*, який може вказувати на об'єкт будь-якого типу, але втрачає інформацію про нього, вимагаючи явного перетворення типу перед використанням.

Ключовими операціями є взяття адреси (&), що дозволяє отримати адресу існуючого об'єкта (ptr = &value;), та розіменування (\*), що надає доступ до значення за адресою, збереженою у вказівнику (data = \*ptr; або \*ptr = newValue;). Важливою концепцією є нульовий вказівник, представлений у сучасному C++ літералом nullptr. Він позначає, що вказівник навмисно не вказує на жоден дійсний об'єкт, і перевірка на nullptr перед розіменуванням є фундаментальним правилом безпечного програмування.

Слід також відрізняти вказівники від посилань C++ (&). Посилання є псевдонімом для існуючого об'єкта і, на відміну від вказівників, мають бути ініціалізовані при оголошенні, не можуть бути нульовими та не можуть бути перепризначені для посилання на інший об'єкт після створення. Посилання часто є простішою та безпечнішою альтернативою для багатьох завдань (особливо передачі параметрів), але вказівники пропонують більшу гнучкість, зокрема можливість бути nullptr, підтримку арифметичних операцій та здатність змінювати адресу, на яку вони вказують.

**1.2 Вказівники та динамічні аспекти C++**

Однією з ключових сфер застосування вказівників є робота з динамічною пам'яттю за допомогою операторів new, delete, new[] та delete[]. Ці оператори дозволяють програмісту запитувати виділення пам'яті з купи (heap) під час виконання програми для об'єктів та масивів, розмір або необхідність яких невідомі на етапі компіляції. new повертає вказівник на початок виділеного блоку пам'яті, який потім використовується для доступу до створеного об'єкта(ів) та для подальшого звільнення пам'яті за допомогою відповідного оператора delete або delete[]. Ручне керування динамічною пам'яттю вимагає суворої дисципліни для уникнення витоків пам'яті та інших помилок.

Арифметика вказівників є ще однією потужною можливістю, тісно пов'язаною з роботою з послідовними даними, особливо з масивами. Оскільки ім'я масиву часто неявно перетворюється на вказівник на його перший елемент, арифметика вказівників (інкремент, декремент, додавання/віднімання цілого числа) надає ефективний спосіб ітерації по елементах масиву. Вираз \*(ptr + i) еквівалентний ptr[i]. Ця властивість є фундаментальною для розуміння того, як функції C/C++ працюють з масивами, отримуючи фактично лише вказівник на їхній початок. Однак, саме тут криється небезпека виходу за межі виділеної пам'яті.

Можливість вказівників зберігати адреси дозволяє створювати динамічні структури даних, такі як зв'язні списки, дерева, графи. В цих структурах елементи (вузли) містять не лише дані, але й один або декілька вказівників на інші елементи. Це дозволяє створювати структури довільного розміру та форми, які можуть ефективно зростати та зменшуватися під час виконання програми, на відміну від статичних масивів фіксованого розміру.

Вказівники також відіграють центральну роль у реалізації поліморфізму в об'єктно-орієнтованому програмуванні на C++. Використання вказівників на базовий клас для посилання на об'єкти похідних класів у поєднанні з віртуальними функціями дозволяє викликати коректну реалізацію методу для фактичного типу об'єкта під час виконання (динамічна диспетчеризація). Це є основою для побудови гнучких та розширюваних систем класів.

**1.3 Перетворення та приведення типів вказівників**

Робота з вказівниками в C++ часто включає операції перетворення їх типів. Ці перетворення можуть бути як неявними, що виконуються компілятором автоматично в певних ситуаціях, так і явними, що вимагають використання спеціальних операторів приведення типу (cast). Розуміння правил та наслідків цих перетворень є критично важливим для написання коректного та безпечного коду[6].

До неявних перетворень належать:

* Перетворення типу масиву на вказівник: Як зазначалося раніше, ім'я масиву в більшості контекстів автоматично перетворюється на вказівник на його перший елемент (T[] до T\*).
* Кваліфікаційні перетворення: Наприклад, вказівник на неконстантний тип (T\*) може бути неявно перетворений на вказівник на константний тип (const T\*). Зворотне перетворення (зняття константності) неявним бути не може.
* Перетворення на void\*: Будь-який вказівник на тип даних (але не на функцію чи член класу) може бути неявно перетворений на void\*. Зворотне перетворення вимагає явного приведення типу.
* Перетворення вказівника на похідний клас до вказівника на базовий клас: Вказівник на об'єкт похідного класу (Derived\*) може бути неявно перетворений на вказівник на публічний базовий клас (Base\*). Це є основою поліморфізму. Зворотне перетворення (від базового до похідного) зазвичай вимагає явного приведення типу та може бути небезпечним, якщо вказівник насправді не вказує на об'єкт похідного класу.
* Перетворення на nullptr: Нульовий вказівник nullptr може бути неявно перетворений на будь-який тип вказівника.

Явне приведення типів вказівників використовується тоді, коли неявне перетворення неможливе або коли потрібно здійснити потенційно небезпечне перетворення. C++ надає чотири оператори для явного приведення типів:

* static\_cast<T\*>: Використовується для "безпечних" перетворень, таких як зворотне перетворення з void\* на оригінальний тип вказівника, або для перетворення між вказівниками на базовий та похідний класи (downcast), коли програміст впевнений у коректності типу. Компілятор виконує певні перевірки під час компіляції.
* reinterpret\_cast<T\*>: Дозволяє інтерпретувати бітове представлення одного вказівника як вказівник іншого, потенційно зовсім не пов'язаного типу. Це найнебезпечніший вид приведення, який слід використовувати вкрай обережно, зазвичай для низькорівневих операцій або взаємодії з апаратним забезпеченням. Він не виконує жодних перевірок безпеки.
* const\_cast<T\*>: Єдиний оператор, що дозволяє зняти const або volatile кваліфікацію з вказівника. Використання const\_cast для модифікації об'єкта, який спочатку був оголошений як const, призводить до невизначеної поведінки. Його основне легітимне застосування — робота зі старими API, які не дотримуються const-коректності.
* C-style cast (T\*)ptr: Успадкований від C синтаксис приведення типів. Він намагається виконати комбінацію static\_cast, const\_cast та reinterpret\_cast і є менш безпечним, оскільки приховує наміри програміста та ускладнює пошук потенційно небезпечних перетворень у коді. Використання C-style cast у C++ коді не рекомендується.

Неправильне використання операторів приведення типів, особливо reinterpret\_cast та const\_cast, може призвести до порушення системи типів, пошкодження пам'яті та невизначеної поведінки.

**1.4 Проблеми безпеки та сучасні практики роботи з вказівниками**

Незважаючи на свою потужність, "сирі" вказівники (raw pointers) є одним із найпоширеніших джерел помилок у програмах на C++. Необхідність ручного керування пам'яттю та гнучкість арифметики вказівників створюють ризики виникнення серйозних проблем. До найпоширеніших належать: витоки пам'яті, коли виділена пам'ять не звільняється; висячі вказівники, що посилаються на вже звільнену або недійсну пам'ять; розіменування нульового вказівника (nullptr); вихід за межі буфера при роботі з масивами через некоректну арифметику вказівників; подвійне звільнення однієї й тієї ж ділянки пам'яті. Ці помилки можуть призводити до непередбачуваної поведінки програми, її аварійного завершення, пошкодження даних та вразливостей безпеки.

Для підвищення безпеки та надійності коду, що працює з вказівниками, важливо дотримуватися певних практик. const-коректність є однією з них. Використання модифікатора const при оголошенні вказівників (const T\*, T\* const, const T\* const) дозволяє на рівні компілятора контролювати, чи можна змінювати дані за вказівником та/або сам вказівник, зменшуючи ймовірність випадкових модифікацій.

Однак ключовим кроком у напрямку безпечного керування ресурсами в сучасному C++ стало введення розумних вказівників (smart pointers), визначених у заголовку <memory>. Вони є класами-обгортками над "сирими" вказівниками, що автоматизують керування життєвим циклом ресурсу (найчастіше — динамічно виділеної пам'яті), реалізуючи принцип RAII (Resource Acquisition Is Initialization). Основні типи розумних вказівників:

* std::unique\_ptr: Забезпечує ексклюзивне володіння ресурсом. Ресурс автоматично звільняється, коли unique\_ptr знищується. Копіювання заборонено, можливе лише переміщення володіння.
* std::shared\_ptr: Реалізує семантику спільного володіння ресурсом за допомогою підрахунку посилань. Ресурс звільняється тоді, коли знищується останній shared\_ptr, що на нього посилався.
* std::weak\_ptr: Неволодіючий вказівник, що використовується разом із shared\_ptr для розриву циклічних посилань, які могли б запобігти звільненню ресурсів.

Використання розумних вказівників замість "сирих" у більшості випадків, де потрібне динамічне керування ресурсами, вважається сучасною найкращою практикою в C++, оскільки воно значно знижує ризик витоків пам'яті та появи висячих вказівників.

**1.5 Методологія дослідження можливостей вказівників**

Дослідження можливостей вказівників у C++ передбачає застосування комбінації методів для отримання повного та об'єктивного розуміння їх функціонування, переваг та недоліків. Теоретичний аналіз включатиме вивчення фундаментальних джерел: стандарту мови C++, класичних підручників (Страуструп, Мейєрс та ін.), сучасної документації та спеціалізованих статей. Це дозволить зрозуміти формальні правила, історичний контекст та еволюцію концепцій, пов'язаних з вказівниками.

Емпіричне дослідження буде реалізовано через створення та аналіз програмних прикладів. Написання коду, що демонструє різні аспекти використання вказівників (арифметика, динамічна пам'ять, робота зі структурами даних, поліморфізм, функціональні вказівники), дозволить спостерігати їхню поведінку на практиці. Важливою частиною цього методу є використання зневаджувачів для покрокового аналізу виконання, інспектування вмісту пам'яті за адресами, які зберігаються у вказівниках, та відстеження їх змін.

Порівняльний аналіз буде застосовано для зіставлення "сирих" вказівників з альтернативними механізмами C++ — посиланнями та розумними вказівниками. Будуть проаналізовані переваги та недоліки кожного підходу в контексті безпеки коду, продуктивності, читабельності та гнучкості для різних завдань. Також можливе коротке порівняння з концепціями вказівників або їх аналогів в інших мовах програмування для кращого розуміння специфіки C++.

Додатково буде проведено аналіз прикладів використання вказівників у реальних системах або бібліотеках, щоб продемонструвати їхню практичну значущість та типові патерни застосування у складних програмних продуктах.

**1.6 Інструментальні засоби для аналізу та роботи з вказівниками**

Ефективне дослідження та робота з вказівниками в C++ потребують використання відповідного інструментарію. Основою є компілятор C++ (GCC, Clang, MSVC), який не тільки перетворює код на виконуваний файл, але й виконує статичний аналіз та може видавати попередження про потенційно небезпечні конструкції з вказівниками (при ввімкнених відповідних опціях, наприклад, -Wall). Системи збірки (CMake, Make) допомагають керувати процесом компіляції складних проектів[7].

Інтегровані середовища розробки (IDE), такі як Visual Studio, CLion, VS Code (з C++ розширеннями), надають єдиний інтерфейс для редагування коду, компіляції, зневадження та навігації. Їхні функції автодоповнення, підсвічування синтаксису та статичного аналізу "на льоту" спрощують розробку та допомагають уникати помилок.

Ключовим інструментом для аналізу поведінки вказівників під час виконання є зневаджувач (debugger) (GDB, LLDB, зневаджувач Visual Studio). Він дозволяє контролювати хід виконання програми, встановлювати точки зупину, переглядати значення змінних (включаючи адреси у вказівниках та дані за цими адресами), аналізувати стек викликів та стан пам'яті.

Для виявлення помилок, пов'язаних із динамічною пам'яттю, які важко знайти за допомогою звичайного зневадження, використовуються інструменти аналізу пам'яті. Valgrind (зокрема, інструмент Memcheck) є потужним засобом для виявлення витоків пам'яті, доступу до неініціалізованої пам'яті, використання звільненої пам'яті та виходу за межі буфера в середовищі Linux/macOS. AddressSanitizer (ASan), MemorySanitizer (MSan), ThreadSanitizer (TSan), які часто інтегруються безпосередньо в компілятори GCC та Clang, надають схожі можливості з меншими накладними витратами під час виконання.

Нарешті, важливим інструментом є доступ до якісної документації та інформаційних ресурсів: офіційний стандарт C++, веб-сайти на кшталт cppreference.com, книги, статті та форуми (Stack Overflow), де можна знайти пояснення, приклади та рішення проблем, пов'язаних з вказівниками.

**2 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

**2.1 Постановка задачі**

Вказівники є однією з найбільш потужних і водночас суперечливих характеристик мови програмування C++. З одного боку, вони надають програмісту безпрецедентний рівень контролю над ресурсами комп'ютера, зокрема над пам'яттю, дозволяючи реалізовувати високоефективні алгоритми, складні структури даних та здійснювати низькорівневу взаємодію з операційною системою та апаратним забезпеченням. З іншого боку, саме вказівники традиційно вважаються одним із найскладніших для опанування аспектів C++ та головним джерелом багатьох поширених помилок, таких як витоки пам'яті, пошкодження даних та вразливості безпеки. Ця двоїста природа вказівників — їхня потужність та пов'язані з нею ризики — визначає центральну проблему, яку ставить перед собою дана курсова робота: необхідність глибокого, структурованого та актуального дослідження можливостей вказівників у C++, їхніх механізмів роботи, сфер застосування, потенційних проблем та сучасних підходів до їх безпечного використання.

**Актуальність та мотивація дослідження**

Актуальність даного дослідження зумовлена кількома факторами. По-перше, незважаючи на еволюцію C++ та появу нових, безпечніших мовних конструкцій (як-от посилання та розумні вказівники), "сирі" вказівники залишаються невід'ємною частиною мови. Вони використовуються у величезній кодовій базі існуючих проектів, є основою для реалізації багатьох стандартних бібліотечних компонентів (включаючи розумні вказівники та контейнери) та залишаються незамінними у певних областях, таких як системне програмування, розробка вбудованих систем та високопродуктивні обчислення, де потрібен максимальний контроль над ресурсами.

По-друге, для студентів та розробників, що вивчають C++, повне розуміння вказівників є критично важливим для досягнення професійного рівня володіння мовою. Поверхневе знайомство лише із синтаксисом недостатньо; необхідно усвідомлювати їхній зв'язок з моделлю пам'яті, принципи роботи арифметики вказівників, механізми динамічного виділення пам'яті та пов'язані з цим відповідальності.

По-третє, помилки, пов'язані з неправильним використанням вказівників, продовжують становити значну загрозу для надійності, безпеки та вартості супроводу програмного забезпечення. Витоки пам'яті можуть призводити до поступової деградації продуктивності та відмови системи. Висячі вказівники та вихід за межі буфера можуть спричинити непередбачувані збої та відкрити шлях для експлойтів безпеки. Вартість виявлення та виправлення таких помилок на пізніх етапах розробки або після релізу продукту може бути надзвичайно високою.

Таким чином, існує постійна потреба у систематизації знань про вказівники, аналізі їхніх можливостей та ризиків у світлі сучасних стандартів C++ (C++11, C++14, C++17, C++20 і далі) та поширених практик розробки. Дана курсова робота має на меті заповнити цю нішу, надавши структурований огляд теми, орієнтований на студентів та розробників, що прагнуть поглибити свої знання про цей фундаментальний аспект C++.

**Основна мета та завдання дослідження**

Основною метою даної курсової роботи є комплексне дослідження концептуальних та практичних можливостей вказівників у мові програмування C++, аналіз пов'язаних із ними ризиків та огляд сучасних підходів до їх безпечного та ефективного використання.

**Межі та обмеження дослідження**

Дана курсова робота зосереджується на дослідженні можливостей вказівників у контексті стандартної мови C++. Хоча будуть згадані основи взаємодії з пам'яттю, робота не заглиблюється в асемблерний рівень представлення вказівників, специфіку апаратної адресації конкретних архітектур, питання вирівнювання даних (data alignment) чи низькорівневі деталі реалізації операційної системи. Також поза межами детального розгляду залишаються деякі більш специфічні або рідко вживані аспекти, такі як вказівники на члени класу (.\*, ->\*), складні патерни кастомних алокаторів пам'яті чи глибока реалізація складних структур даних. Основний фокус роботи — на ядрі концепції вказівників, їхніх ключових можливостях, проблемах безпеки та сучасних практиках використання, що є найбільш актуальним для студентів та розробників, які вивчають C++. Дослідження орієнтоване на стандарт C++ (з акцентом на C++11 та новіші версії, де були введені nullptr та розумні вказівники) і не розглядає нестандартні розширення компіляторів.

**Очікувані результати**

За підсумками виконання даної курсової роботи очікується отримання наступних результатів:

* Систематизований опис механізмів роботи вказівників у C++, включаючи їхній синтаксис, семантику, арифметику, перетворення типів та взаємодію з const.
* Глибокий аналіз ролі вказівників у керуванні динамічною пам'яттю та пов'язаних із цим поширених помилок.
* Аналіз застосування вказівників для реалізації важливих програмних конструкцій, таких як динамічні структури даних, поліморфізм та механізми зворотного виклику.
* Порівняльний аналіз "сирих" вказівників та сучасних розумних вказівників (std::unique\_ptr, std::shared\_ptr, std::weak\_ptr) з точки зору безпеки, ефективності та практик програмування.
* Набір ілюстративних програмних прикладів, що демонструють ключові можливості та проблеми використання вказівників.
* Обґрунтовані висновки та рекомендації щодо безпечного та ефективного використання різних типів вказівників у сучасній розробці на C++.

Результати роботи будуть представлені у вигляді структурованого текстового звіту, що містить теоретичний аналіз, опис практичних прикладів та висновки, і можуть бути корисними для студентів, які вивчають C++, а також для розробників, що прагнуть освіжити або поглибити свої знання про вказівники.

**2.1.1 Завдання до КР**

Реалізація мети даної курсової роботи, що полягає у дослідженні можливостей вказівників у C++, потребувала не лише теоретичного аналізу, але й практичної демонстрації ключових концепцій. Тому невід'ємною частиною дослідження стало виконання низки конкретних завдань у рамках розробки програмного прикладу. Цей приклад слугує експериментальним майданчиком для ілюстрації поведінки вказівників, роботи з динамічною пам'яттю та взаємодії з іншими елементами мови. Даний підрозділ детально описує завдання, що були поставлені перед програмною реалізацією, та розкриває, які саме аспекти функціонування вказівників вона покликана продемонструвати.

Першочерговим завданням було **продемонструвати фундаментальний синтаксис та базові операції,** пов'язані з "сирими" вказівниками. Розроблений код ілюструє процес оголошення вказівника на користувацький тип даних (struct Data), що є типовим сценарієм у C++. Особлива увага приділена ініціалізації вказівника: показано як отримання адреси динамічно виділеного об'єкта за допомогою оператора new, так і явну ініціалізацію нульовим значенням за допомогою nullptr. Було продемонстровано використання оператора доступу до членів структури через вказівник (->) як для читання (raw\_data\_ptr->value), так і для запису (raw\_data\_ptr->value = ...). Крім того, код включає перевірку вказівника на рівність nullptr (if (null\_ptr == nullptr)), підкреслюючи важливість таких перевірок перед потенційним розіменуванням для уникнення помилок часу виконання. Це завдання формує базове розуміння того, як вказівники оголошуються, як їм присвоюються адреси, як через них отримувати доступ до даних, та як безпечно позначати вказівники, що не вказують на дійсний об'єкт.

Наступним важливим завданням стала **реалізація та демонстрація ручного керування динамічною пам'яттю**. Оскільки вказівники нерозривно пов'язані з можливістю виділяти пам'ять під час виконання програми, код мав чітко показати цей процес. Було реалізовано виділення пам'яті для одиночного об'єкта типу Data за допомогою оператора new. Приклад показує, що new повертає адресу виділеного блоку пам'яті, яка зберігається у відповідному вказівнику (raw\_data\_ptr). Не менш важливим було показати процес звільнення цієї пам'яті. Код містить явний виклик оператора delete для вказівника raw\_data\_ptr наприкінці його використання. Щоб візуалізувати життєвий цикл об'єкта в динамічній пам'яті, структура Data була доповнена конструктором та деструктором, які виводять відповідні повідомлення в консоль при створенні та знищенні об'єкта. Це дозволяє спостерігачеві бачити момент виділення пам'яті (виклик конструктора) та момент її звільнення (виклик деструктора), підкреслюючи відповідальність програміста за своєчасне та коректне звільнення ресурсів при ручному керуванні.

Для порівняння та демонстрації сучасних підходів було поставлено завдання **показати автоматичне керування динамічною пам'яттю за допомогою розумних вказівників**, зокрема std::unique\_ptr. Код мав продемонструвати реалізацію патерну RAII (Resource Acquisition Is Initialization). Було створено об'єкт Data за допомогою new, але отриманий "сирий" вказівник був негайно переданий в конструктор std::unique\_ptr<Data>. Це ілюструє захоплення ресурсу (динамічної пам'яті) в момент ініціалізації об'єкта-обгортки (unique\_ptr). Подальший доступ до членів об'єкта Data через unique\_data\_ptr здійснюється так само, як і через "сирий" вказівник (оператор ->), показуючи, що розумні вказівники імітують інтерфейс звичайних вказівників. Ключова відмінність, яку мав продемонструвати код, — це відсутність явного виклику delete. Відповідне повідомлення від деструктора об'єкта Data, керованого unique\_ptr, що з'являється при завершенні функції main (коли unique\_ptr виходить з області видимості), наочно показує автоматичне звільнення ресурсу. Це завдання підкреслює переваги RAII та розумних вказівників у запобіганні витокам пам'яті та спрощенні коду.

Ще одним завданням стала **реалізація роботи з вказівниками на функції**. Ця можливість C++ дозволяє зберігати адреси функцій та викликати їх непрямо. Програмний приклад включає оголошення вказівника operation\_ptr, сумісного за сигнатурою з функцією add (int (\*)(int, int)). Було показано процес присвоєння адреси функції add цьому вказівнику. Далі, для демонстрації непрямого виклику, програма отримує від користувача два числа та викликає операцію (в даному випадку, додавання) саме через вказівник operation\_ptr. Це завдання ілюструє механізм, що лежить в основі таких патернів, як зворотні виклики (callbacks) або стратегії, де конкретна дія може визначатися динамічно.

Важливим завданням було також **продемонструвати роботу з динамічним масивом вказівників**, що вимагає розуміння вказівників на вказівники (Data\*\*). Код реалізує сценарій, де спочатку за запитом користувача визначається розмір масиву, а потім за допомогою new Data\*[array\_size] виділяється пам'ять під сам масив, який буде зберігати адреси об'єктів Data. Потім у циклі відбувається заповнення цього масиву: для кожного елемента масиву в динамічній пам'яті створюється окремий об'єкт Data (знову за допомогою new), і вказівник на цей об'єкт зберігається у відповідній комірці масиву (array\_of\_pointers[i]). Доступ до даних об'єктів демонструється через індексацію масиву вказівників та оператор ->. Критично важливим аспектом цього завдання була демонстрація **правильного двоетапного звільнення пам'яті**. Код явно показує, що спочатку необхідно в циклі пройти по масиву і викликати delete для кожного об'єкта Data, на який вказували елементи масиву, і лише після цього можна звільняти пам'ять, виділену для самого масиву вказівників, за допомогою delete[] array\_of\_pointers. Це завдання підкреслює складнощі ручного керування пам'яттю при роботі з більш комплексними структурами даних.

Крім основних механізмів, було поставлено завдання **ілюструвати використання кваліфікатора** const **з вказівниками та роботу з** void\*. Код включає приклади оголошення та використання вказівника на константу (const int\*) та константного вказівника (int\* const), показуючи, які операції (зміна даних за адресою чи зміна самої адреси) дозволені чи заборонені для кожного випадку. Це демонструє, як const-коректність може підвищити безпеку коду. Також було показано використання void\* як універсального вказівника, що може зберігати адресу будь-якого об'єкта. Приклад ілюструє присвоєння адреси Data\* вказівнику void\* та наступне безпечне зворотне перетворення типу за допомогою static\_cast<Data\*> перед доступом до даних, підкреслюючи необхідність явного приведення типів при роботі з void\*[4].

Нарешті, окремим завданням було **забезпечити інтерактивність прикладу та візуалізацію результатів**. Програма використовує стандартні потоки введення/виведення (std::cin, std::cout) та допоміжну функцію readInt для отримання даних від користувача (значень, розмірів масивів), що робить демонстрацію більш гнучкою. Виведення в консоль повідомлень про хід виконання, результати операцій, а також повідомлень від конструкторів/деструкторів об'єктів Data дозволяє користувачеві спостерігати за процесами виділення, використання та звільнення ресурсів, керованих вказівниками.

Таким чином, комплекс завдань, реалізованих у програмному прикладі, охоплює широкий спектр фундаментальних операцій та типових сценаріїв використання вказівників у C++, від базового синтаксису та ручного керування пам'яттю до автоматичного керування за допомогою std::unique\_ptr, роботи з масивами вказівників, вказівниками на функції та аспектами типізації й const-коректності. Виконання цих завдань забезпечує наочну практичну основу для розуміння матеріалу, що викладається в теоретичних розділах даної курсової роботи.

#### 2.1.2 Вхідна інформація

Програмний додаток, розроблений для керування списком студентів та їхніх оцінок, функціонує в інтерактивному режимі, покладаючись на дані, що вводяться користувачем через стандартний консольний потік введення (std::cin). Ці вхідні дані визначають логіку роботи програми на кожному кроці та дозволяють динамічно наповнювати й переглядати інформаційну базу студентів. Процес введення супроводжується текстовими підказками для користувача. Нижче детально розглянуто кожен тип вхідної інформації, що очікується програмою:

**1. Вибір опції меню -** це основний керуючий елемент програми. Після виведення текстового меню, яке перелічує доступні операції (1: Додати студента та оцінку, 2: Показати список студентів, 0: Вихід), програма виводить запит: "Ваш вибір: ". Користувач повинен ввести числове значення, що відповідає бажаній дії. Для зчитування цього значення використовується функція readDouble, яка повертає тип double, але потім результат явно приводиться до int для використання в операторі switch. Такий підхід до зчитування (через readDouble) обрано для уніфікації використання функції перевірки введення. Введене число визначає, яка гілка switch буде виконана: додавання нового запису, відображення наявних записів, завершення роботи програми або обробка невірного вибору. Цей запит повторюється в циклі do-while доти, доки користувач не введе 0 для виходу (приклад відображення меню та запиту вибору наведено на Рисунку 2.1).

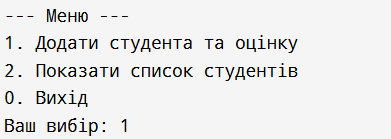


Рисунок 2.1 – Меню програми

**2. Ім'я студента -** цей тип вхідних даних запитується лише у випадку, коли користувач обрав опцію "1" (Додати студента та оцінку). Програма виводить текстовий запит: "Введіть ім'я студента: " (див. Рисунок 2.2). Очікується введення текстового рядка, який може містити літери, пробіли та інші символи, що формують повне ім'я студента. Для зчитування цього рядка використовується спеціалізована функція readString, яка використовує std::getline(std::cin >> std::ws, value) для коректного зчитування цілого рядка, включаючи пробіли після можливих залишкових символів нового рядка у буфері введення (завдяки маніпулятору std::ws). Отриманий рядок зберігається безпосередньо у полі name об'єкта new\_student типу Student. Це введення є критично важливим для ідентифікації запису про студента у списку.

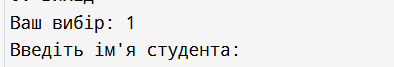


Рисунок 2.2 – Ввід ім’я студента

**3. Оцінка студента:** Слідом за введенням імені (якщо було обрано опцію "1"), програма запитує числове значення оцінки за допомогою текстового запиту: "Введіть оцінку студента: " (див. Рисунок 2.3). Користувач має ввести число, яке може бути цілим або дробовим, оскільки для зберігання оцінки використовується тип double. Використовується функція readDouble для зчитування значення типу double, яке потім зберігається у полі grade об'єкта new\_student. Успішно зчитане числове значення представляє кількісний показник успішності студента.

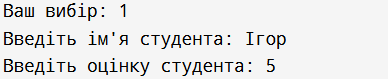


Рисунок 2.3 – Ввід оцінки студента

**Я**к було зазначено, для зчитування числових даних (вибір меню, оцінка) та рядків (ім'я) використовуються допоміжні функції readDouble та readString. Функція readDouble додатково реалізує цикл перевірки стану потоку введення (std::cin.good()). Якщо стан потоку некоректний (наприклад, через введення літер замість цифр), функція очищає прапорці помилок (std::cin.clear()), ігнорує некоректні символи у буфері (std::cin.ignore(...)), виводить повідомлення про помилку та повторює запит на введення (приклад обробки помилкового введення показано на Рисунку 2.4). Це забезпечує стійкість програми до непередбачуваного або помилкового введення даних користувачем.

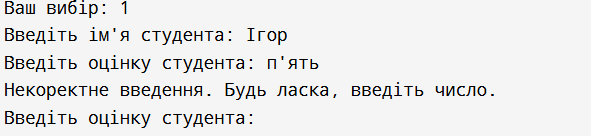


Рисунок 2.4 – Некоректні вхідні дані

Узагальнюючи, вхідна інформація для програми керування списком студентів складається з команд користувача (вибір меню) та атрибутів даних (ім'я, оцінка), що дозволяють маніпулювати списком. Інтерактивний характер введення та наявність базової перевірки коректності роблять програму функціональною для виконання поставлених завдань у рамках її простої логіки.

#### 2.1.3 Вихідна інформація

Результатом роботи програми керування списком студентів є текстова інформація, що виводиться у стандартний потік виведення (консоль). Ця вихідна інформація слугує для кількох цілей: керування взаємодією з користувачем (відображення меню та запитів на введення), інформування про результати виконаних операцій (додавання студента, відображення списку), повідомлення про стан програми (порожній список, помилки введення) та підтвердження завершення роботи. Структура та зміст вихідної інформації розроблені таким чином, щоб забезпечити зрозумілий та послідовний інтерфейс користувача.

1. Відображення головного меню та запиту вибору - основою взаємодії є текстове меню, яке програма виводить на початку кожної ітерації основного циклу do-while. Це статичний блок тексту, що інформує користувача про доступні опції (лістинг 2.1):

Лістинг 2.1 – Виведення меню

std::cout << "\n--- Меню ---" << std::endl;

std::cout << "1. Додати студента та оцінку" << std::endl;

std::cout << "2. Показати список студентів" << std::endl;

std::cout << "0. Вихід" << std::endl;

Одразу після меню виводиться запит на введення команди: "Ваш вибір: ". Цей блок вихідної інформації є незмінним і слугує для навігації користувача по функціоналу програми. Його чітке та лаконічне представлення є важливим для зручності використання (див. Рисунок 2.1, де показано вигляд меню та запиту).

2. Запити на введення даних студента - у випадку вибору опції "1" (Додати студента), програма генерує специфічну вихідну інформацію у вигляді послідовних запитів на введення даних:

* "Введіть ім'я студента: " - запит на введення текстового рядка.
* "Введіть оцінку студента: " - запит на введення числового значення (оцінки). Ці текстові повідомлення є ключовими для отримання вхідних параметрів, необхідних для створення нового запису про студента (приклади запитів можна побачити на Рисунках 2.2 та 2.3, що ілюструють процес введення даних).

3. Повідомлення про успішне додавання - після того, як користувач ввів ім'я та оцінку для нового студента, програма підтверджує успішне завершення операції додавання, виводячи коротке інформаційне повідомлення: "Студента додано.\n". Це повідомлення слугує зворотним зв'язком для користувача, інформуючи його про те, що введені дані були оброблені та відповідний запис було внесено до внутрішньої структури даних (вектора student\_list).

4. Відображення списку студентів - при виборі опції "2" програма генерує вихідну інформацію, що відображає поточний вміст списку студентів. Формат цього виводу залежить від наявності записів у списку:

* Якщо список порожній - програма виводить єдине повідомлення: "Список студентів порожній.\n". Це інформує користувача про поточний стан бази даних (рисунок 2.5).

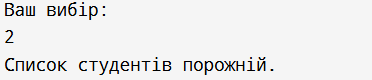


Рисунок 2.5 – Повідомлення про порожній список студентів

* Якщо список містить записи - програма спочатку виводить заголовок "\n--- Список студентів ---", а потім у циклі проходить по всіх елементах вектора student\_list. Для кожного студента s у векторі генерується рядок виводу у форматі: "Ім'я: [ім'я студента], Оцінка: [оцінка студента]\n", де [ім'я студента] замінюється значенням s.name, а [оцінка студента] – значенням s.grade. Дані кожного студента виводяться на окремому рядку. Цей форматований вивід є основним способом представлення результатів роботи програми – збережених даних про студентів. Він дозволяє користувачеві перевірити коректність раніше введених даних та побачити поточний стан списку (приклад виводу списку з кількома студентами наведено на Рисунку 2.6). Чітке форматування забезпечує легкість читання та інтерпретації представленої інформації.

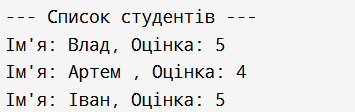


Рисунок 2.6 - Список студентів

5. Повідомлення про помилки - програма генерує специфічну вихідну інформацію у відповідь на помилкові дії користувача:

* Помилка введення числових даних - якщо при введенні вибору меню або оцінки користувач вводить нечислові символи, функція readDouble генерує повідомлення: "Некоректне введення. Будь ласка, введіть число.\n". Після цього повідомлення програма автоматично повторює запит на введення, дозволяючи користувачеві виправити помилку (Рисунок 2.7).

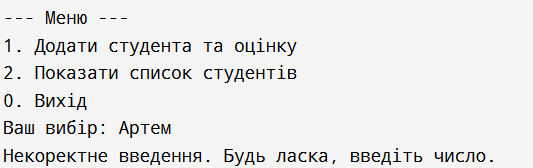


Рисунок 2.7 – Некоректний вибір меню (строка)

* Некоректний вибір пункту меню **-**  якщо користувач вводить число, яке не відповідає жодному з пунктів меню (не 1, 2 або 0), виконується гілка default оператора switch, яка виводить повідомлення: "Невірний вибір. Спробуйте ще раз.\n" (Рисунок 2.8). Після цього програма повертається до відображення головного меню та запиту вибору. Ці повідомлення про помилки є важливою частиною вихідної інформації, оскільки вони надають користувачеві зворотний зв'язок про некоректність його дій та спрямовують його до правильного використання програми.

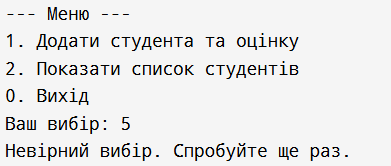


Рисунок 2.8 – Некоректний вибір меню (число)

6.Повідомлення про завершення роботи - при виборі опції "0" (Вихід) програма виводить фінальне повідомлення: "Вихід з програми.\n", після чого її виконання завершується (Рисунок 2.9). Це стандартне повідомлення, що інформує користувача про успішне завершення роботи за його командою.

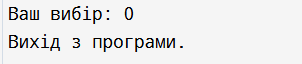


Рисунок 2.9 – Вихід з програми

Таким чином, вихідна інформація програми є набором текстових повідомлень, що включають статичне меню, динамічні запити на введення, підтвердження операцій, форматований вивід даних зі списку студентів, а також повідомлення про помилки та завершення роботи. Ця інформація формує консольний інтерфейс користувача програми та відображає результати її роботи і поточний стан внутрішніх даних (списку студентів). Якість та зрозумілість цієї вихідної інформації безпосередньо впливають на зручність та ефективність використання програми.

### 2.2 Опис мови програмування

Мова програмування C++, що є об'єктом вивчення у даній курсовій роботі та мовою реалізації практичних прикладів, є однією з найвпливовіших та найпоширеніших мов програмування у світі. Створена Б'ярном Страуструпом на початку 1980-х років як розширення мови C, C++ була задумана з метою додати до C можливості об'єктно-орієнтованого програмування (ООП) та інші вдосконалення, зберігаючи при цьому високу продуктивність, ефективність та гнучкість свого попередника. З моменту свого створення C++ пройшла значний шлях розвитку, що було закріплено у міжнародних стандартах ISO/IEC: C++98, C++03, C++11, C++14, C++17, C++20 та C++23. Кожен новий стандарт додавав до мови нові можливості, бібліотеки та парадигми програмування, роблячи її більш виразною, безпечною та потужною[5].

C++ характеризується як мультипарадигменна мова програмування. Вона підтримує процедурний підхід (успадкований від C), об'єктно-орієнтований (класи, спадкування, поліморфізм), узагальнений (шаблони) та, все більшою мірою, функціональний стилі програмування. Це компільована мова, тобто вихідний код перетворюється у машинний код специфічної платформи перед виконанням, що забезпечує високу швидкість роботи програм. C++ є статично типізованою мовою, де типи всіх змінних перевіряються на етапі компіляції, що допомагає виявляти багато помилок заздалегідь. Однією з ключових особливостей C++, що є центральною темою даної роботи, є можливість низькорівневого доступу до пам'яті через механізм вказівників, що дозволяє створювати високоефективні програми з мінімальними накладними витратами[1].

Завдяки поєднанню високої продуктивності, гнучкості та можливості низькорівневого контролю, C++ займає домінуючі позиції у багатьох галузях розробки програмного забезпечення, включаючи системне програмування (операційні системи, драйвери), розробку ігрових рушіїв та відеоігор ААА-класу, високочастотну торгівлю (HFT), розробку вбудованих систем, наукові та інженерні обчислення, комп'ютерну графіку, а також створення інфраструктурних компонентів для інших систем. Водночас, потужність C++ пов'язана зі значною складністю самої мови та підвищеними вимогами до кваліфікації розробника, особливо у питаннях керування ресурсами.

**Ключові мовні засоби C++**

Розуміння можливостей вказівників неможливе без знання основних елементів мови C++, з якими вони взаємодіють[3]:

* Система типів - C++ має багату систему типів даних. Фундаментальні типи включають цілочисельні (int, char, short, long, long long, bool), типи з плаваючою комою (float, double, long double). Модифікатори signed та unsigned дозволяють керувати діапазоном та представленням цілочисельних типів. Розмір та представлення цих типів у пам'яті (що можна дізнатися за допомогою оператора sizeof) є платформо-залежними, що важливо враховувати при роботі з вказівниками та пам'яттю напряму. Окрім фундаментальних, C++ дозволяє створювати користувацькі типи даних за допомогою структур (struct), класів (class) та перелічень (enum). Вказівники можуть посилатися на об'єкти будь-якого з цих типів.
* Змінні, пам'ять та область видимості - змінні в C++ мають тип, ім'я, адресу в пам'яті, час життя (storage duration) та область видимості (scope). За часом життя змінні поділяються на автоматичні (локальні змінні на стеку, існують до виходу з блоку коду), статичні (глобальні змінні, статичні члени класів, локальні статичні змінні, існують протягом усього виконання програми) та динамічні (об'єкти, створені в купі за допомогою new, час життя яких контролюється програмістом). Область видимості визначає, де в коді можна отримати доступ до змінної (локальна, простору імен, класу, глобальна). Ці концепції тісно пов'язані з моделлю пам'яті (стек, купа, статична область).
* Оператори - C++ підтримує широкий набір операторів: арифметичні (+, -, \*, /, %), порівняння (==, !=, <, >, <=, >=), логічні (&&, ||, !), побітові (&, |, ^, ~, <<, >>), присвоєння (=, +=, -=, ...). Особливе значення для теми вказівників мають оператори: розіменування (\*), взяття адреси (&), доступу до членів через вказівник (->), виділення та звільнення динамічної пам'яті (new, delete, new[], delete[]), а також індексації ([]), який може застосовуватися як до масивів, так і до вказівників.
* Керуючі конструкції - мова надає стандартний набір конструкцій для керування потоком виконання: умовні оператори (if, else if, else), оператор вибору (switch), цикли (for, while, do-while, range-based for loop), оператори переходу (break, continue, goto, return).
* Функції є основними будівельними блоками програм. Вони можуть приймати параметри (за значенням, за посиланням (&) або за вказівником (\*)), повертати значення (включаючи вказівники або посилання) та бути перевантаженими (кілька функцій з однаковим ім'ям, але різними списками параметрів). Розуміння механізму виклику функцій та передачі параметрів важливе при роботі з вказівниками на функції та при передачі вказівників у функції.

**Підтримка об'єктно-орієнтованого програмування (ООП)**

ООП є однією з центральних парадигм, підтримуваних C++.

* Класи та об'єкти - клас (class) визначає користувацький тип даних, що об'єднує дані (члени даних) та функції для їх обробки (методи або члени-функції). Об'єкт є екземпляром класу. Класи підтримують інкапсуляцію – приховування внутрішньої реалізації та захист даних за допомогою специфікаторів доступу (public, protected, private). Конструктори використовуються для ініціалізації об'єктів при їх створенні, а деструктори – для звільнення ресурсів перед знищенням об'єкта (особливо важливо для класів, що керують динамічною пам'яттю). Кожен нестатичний метод класу неявно отримує вказівник this на об'єкт, для якого він був викликаний.
* Спадкування **-** механізм, що дозволяє створювати нові класи (похідні) на основі існуючих (базових), успадковуючи їхні властивості та поведінку. C++ підтримує як одиничне, так і множинне спадкування. Спадкування є основою для повторного використання коду та побудови ієрархій класів.
* Поліморфізм - здатність об'єктів різних типів реагувати на один і той же виклик методу по-різному. C++ підтримує статичний поліморфізм (через перевантаження функцій та шаблони) та динамічний поліморфізм (через віртуальні функції). Динамічний поліморфізм дозволяє працювати з об'єктами похідних класів через вказівники або посилання на базовий клас, при цьому виклик віртуальної функції призводить до виконання реалізації, що відповідає фактичному типу об'єкта під час виконання. Саме вказівники є ключовим механізмом реалізації динамічного поліморфізму.

**Керування ресурсами та сучасний C++**

Історично керування ресурсами (особливо динамічною пам'яттю) в C++ було ручним і схильним до помилок. Сучасні стандарти C++ пропагують підходи, що підвищують безпеку та надійність.

* Принцип RAII (Resource Acquisition Is Initialization) - ресурс (пам'ять, файл, м'ютекс тощо) захоплюється в конструкторі об'єкта-обгортки, а звільняється автоматично в деструкторі цього об'єкта, коли він виходить з області видимості. Це гарантує звільнення ресурсу навіть у випадку виникнення винятків.
* Розумні вказівники **-** класи std::unique\_ptr, std::shared\_ptr, std::weak\_ptr реалізують принцип RAII для динамічно виділеної пам'яті, надаючи різні семантики володіння (ексклюзивне, спільне, слабке) та автоматизуючи виклик delete/delete[]. Їх використання є рекомендованою практикою замість "сирих" вказівників для керування динамічними ресурсами.
* Семантика переміщення (Move Semantics) - введена в C++11 (&&, std::move), вона дозволяє ефективно передавати володіння ресурсами між об'єктами без дорогого копіювання, що особливо важливо для ресурсів, якими керують std::unique\_ptr.
* Обробка винятків - механізм try-catch-throw дозволяє обробляти помилки часу виконання. Правильне керування ресурсами за допомогою RAII (розумних вказівників) є критично важливим для написання коду, безпечного щодо винятків (exception-safe code), оскільки гарантує звільнення ресурсів при "розкручуванні стека" під час обробки винятку.

**Стандартна бібліотека**

Окрім ядра мови, C++ має велику Стандартну бібліотеку, яка надає широкий набір інструментів: вже згадану STL (контейнери, ітератори, алгоритми), потоки вводу-виводу (<iostream>, <fstream>), утиліти для роботи з рядками (<string>), засоби для роботи з пам'яттю (<memory>), інструменти для багатопотоковості (<thread>, <mutex>) та багато іншого. Використання стандартної бібліотеки дозволяє писати більш надійний, портативний та ефективний код, часто абстрагуючись від низькорівневих деталей, таких як пряма маніпуляція пам'яттю через вказівники (наприклад, std::vector керує пам'яттю для своїх елементів автоматично).

**Процес компіляції**

Програма на C++ проходить кілька етапів перед виконанням. Препроцесор обробляє директиви (наприклад, #include, #define). Компілятор перетворює оброблений вихідний код на об'єктні файли, що містять машинний код та інформацію для лінкера. Лінкер (компонувальник) об'єднує один або декілька об'єктних файлів та необхідні бібліотеки в єдиний виконуваний файл. Розуміння цього процесу важливе для організації коду (розподіл на заголовні .h/.hpp та файли реалізації .cpp) та керування залежностями[2].

### 2.3 Опис алгоритму та ідентифікатори

Даний підрозділ присвячений опису алгоритму роботи програмного додатку, розробленого для керування списком студентів, а також визначенню основних ідентифікаторів (змінних, функцій, типів даних), що використовуються в його реалізації. Розуміння алгоритму та призначення ідентифікаторів є ключовим для аналізу логіки програми та її функціональності. Повний лістинг програмного коду, що реалізує описаний алгоритм, наведено у **Додатку А**. Візуальне представлення алгоритму у вигляді блок-схеми міститься у **Додатку Б**.

**Опис алгоритму програми**

Програма реалізує простий консольний інтерфейс для взаємодії з користувачем та виконує циклічний процес обробки команд до моменту явного вибору опції виходу. Основний алгоритм роботи програми можна описати наступними кроками:

1. Ініціалізація **-** на початку виконання функції main створюється порожній динамічний масив (вектор) student\_list для зберігання об'єктів типу Student та оголошується змінна choice для зберігання вибору користувача.
2. Основний цикл програми (do-while) - програма входить у цикл, який продовжується доти, доки користувач не обере опцію виходу (значення choice не стане рівним 0).
3. Відображення меню - на кожній ітерації циклу програма виводить у консоль текстове меню з доступними опціями: "1. Додати студента та оцінку", "2. Показати список студентів", "0. Вихід".
4. Отримання вибору користувача - програма виводить запит "Ваш вибір: " і очікує на введення числового значення від користувача. Введене значення зчитується за допомогою функції readDouble, перевіряється на коректність і зберігається (після приведення до int) у змінній choice.
5. Обробка вибору (switch) - залежно від значення змінної choice, виконується одна з наступних дій:
   * Case 1 (Додати студента):
     + Створюється локальний об'єкт new\_student типу Student.
     + За допомогою функції readString зчитується ім'я студента, введене користувачем, і записується у поле new\_student.name.
     + За допомогою функції readDouble зчитується оцінка студента, введена користувачем, і записується у поле new\_student.grade.
     + Створений об'єкт new\_student додається в кінець вектора student\_list за допомогою методу push\_back(). std::vector автоматично керує виділенням необхідної пам'яті.
     + Виводиться підтверджуюче повідомлення "Студента додано.\n".
   * Case 2 (Показати список):
     + Перевіряється, чи вектор student\_list порожній, за допомогою методу empty().
     + Якщо вектор порожній, виводиться повідомлення "Список студентів порожній.\n".
     + Якщо вектор не порожній, виводиться заголовок "\n--- Список студентів ---". Потім програма ітерує по всіх елементах вектора student\_list за допомогою range-based for циклу (for (const Student& s : student\_list)). Для кожного елемента s (що є константним посиланням на об'єкт Student у векторі) виводиться його ім'я (s.name) та оцінка (s.grade) у форматованому рядку.
   * Case 0 (Вихід):
     + Виводиться повідомлення "Вихід з програми.\n". Умова циклу do-while ( choice != 0 ) стає хибною, і виконання циклу та програми завершується.
   * Default (Невірний вибір):
     + Якщо введене значення choice не дорівнює 0, 1 або 2, виводиться повідомлення про помилку: "Невірний вибір. Спробуйте ще раз.\n".
6. Повторення циклу **-** якщо не було обрано опцію виходу (0), програма повертається до кроку 3 (відображення меню).

Блок-схема алгоритму детально візуалізує описану послідовність кроків та умовних переходів і наведена у **Додатку Б**.

**Опис основних ідентифікаторів**

Для реалізації описаного алгоритму в програмному коді використовуються наступні ключові ідентифікатори:

* **Типи даних:**
  + Student: Структура, що визначає користувацький тип даних для зберігання інформації про одного студента. Містить два публічних члени:
    - name (тип std::string): Зберігає ім'я студента.
    - grade (тип double): Зберігає оцінку студента.
* **Функції:**
  + readDouble(const char\* prompt): Допоміжна функція, призначена для безпечного зчитування значення типу double з консолі. Приймає як аргумент текстовий рядок-запрошення (const char\*). Повертає зчитане значення double. Містить внутрішній цикл для обробки помилок введення.
  + readString(const char\* prompt): Допоміжна функція для зчитування рядка (std::string) з консолі, включаючи пробіли. Приймає текстовий рядок-запрошення (const char\*). Повертає зчитаний рядок std::string.
  + main(): Головна функція програми, з якої починається виконання. Вона містить основний цикл програми, логіку обробки меню та виклики інших функцій. Повертає 0 у разі успішного завершення.
* **Змінні (у функції main):**
  + student\_list: Об'єкт типу std::vector<Student>. Динамічний масив (вектор), що зберігає об'єкти типу Student. Слугує основним сховищем даних програми. std::vector автоматично керує виділенням та звільненням пам'яті для своїх елементів.
  + choice: Змінна типу int, що зберігає вибір пункту меню, введений користувачем. Визначає, яка гілка оператора switch буде виконана.
  + new\_student: Локальна змінна типу Student у гілці case 1. Використовується для тимчасового зберігання даних нового студента перед додаванням у вектор student\_list.
  + s: Змінна циклу у range-based for (case 2). На кожній ітерації є константним посиланням (const Student&) на черговий елемент вектора student\_list.
* **Бібліотеки:**
  + iostream: Надає засоби для роботи зі стандартними потоками введення/виведення (std::cin, std::cout, std::endl).
  + string: Надає клас std::string для роботи з текстовими рядками та функцію std::to\_string (хоча остання не використовується у фінальній версії цього коду, але була в попередніх).
  + vector: Надає шаблон класу std::vector для роботи з динамічними масивами.
  + limits: Використовується для доступу до властивостей числових типів (std::numeric\_limits), зокрема, для очищення буфера введення у функції readDouble.
  + ios: Надає базові засоби для роботи з потоками, включаючи маніпулятор std::ws, що використовується у readString.

Повний лістинг програмного коду, що містить усі описані ідентифікатори та реалізує зазначений алгоритм, представлено у **Додатку А**.

**2.4 Тестування програми**

Після розробки програмного додатку для керування списком студентів було проведено етап функціонального тестування з метою перевірки коректності його роботи, стабільності та відповідності очікуваній поведінці. Основна мета полягала у тому, щоб переконатися, що програма правильно виконує заявлені функції: додавання записів про студентів, відображення списку та обробку вибору користувача через меню, а також адекватно реагує на можливі помилки введення.

Тестування проводилося вручну, шляхом безпосередньої взаємодії з програмою через консольний інтерфейс у типовому для розробки середовищі ([Вкажіть ваше ОС та компілятор]). Стратегія тестування поєднувала підхід "чорної скриньки", де перевірялася відповідність реакції програми на певні вхідні дані очікуваним результатам, з елементами "білої скриньки" для цілеспрямованої перевірки обробки різних гілок алгоритму (наприклад, умовних операторів та оператора switch).

У ході тестування було перевірено основні сценарії використання програми. По-перше, була протестована **коректність роботи з порожнім списком**: при виборі опції відображення одразу після запуску програма коректно виводила повідомлення про відсутність записів (приклад виводу для порожнього списку див. на Рисунку 2.5). По-друге, була ретельно перевірена **функція додавання студентів**: тестувалося додавання як одного, так і декількох студентів поспіль. При цьому контролювалося коректне зчитування імені (включаючи імена з пробілами, що обробляються std::getline) та оцінки (включаючи дробові значення). Після кожного успішного введення даних програма виводила підтверджуюче повідомлення "Студента додано.\n", що відповідало очікуванням. По-третє, перевірялася **функція відображення списку** після додавання одного або кількох записів. Спостереження підтвердили, що програма коректно виводить заголовок списку та послідовно відображає дані всіх доданих студентів у форматі "Ім'я: [ім'я], Оцінка: [оцінка]", зберігаючи порядок їх додавання (приклад виводу списку з кількома записами наведено на Рисунку 2.6).

Окрема увага приділялася **тестуванню обробки помилок та некоректних дій користувача**. Було перевірено реакцію програми на введення нечислових даних у відповідь на запити, що очікують число (вибір меню, оцінка). У цих випадках програма коректно виводила повідомлення "Некоректне введення. Будь ласка, введіть число.\n" та повторно запитувала введення, що свідчить про належну роботу функції валідації readDouble (приклад обробки помилки введення див. на Рисунку 2.4). Також було протестовано введення некоректного номера пункту меню (наприклад, 5 або -1). Програма адекватно реагувала, виводячи повідомлення "Невірний вибір. Спробуйте ще раз.\n" і повертаючись до відображення головного меню без збоїв у роботі. Нарешті, була перевірена **коректність завершення роботи** програми при виборі опції "0", що супроводжувалося виведенням повідомлення "Вихід з програми.\n".

За результатами проведеного ручного тестування можна зробити висновок, що розроблений програмний додаток функціонує коректно відповідно до описаного алгоритму. Основні функції додавання та відображення записів працюють належним чином, програма адекватно обробляє як порожній список, так і список з кількома елементами. Реалізовано базову обробку помилок некоректного введення даних та вибору пунктів меню. У всіх тестових сценаріях фактична поведінка програми відповідала очікуваній. Хоча проведене тестування не може гарантувати відсутність абсолютно всіх можливих дефектів, воно покриває основні сценарії використання та підтверджує працездатність реалізованої функціональності.

## ВИСНОВКИ

У даній курсовій роботі було проведено дослідження можливостей вказівників у мові програмування C++. Мета полягала в аналізі їх функціонування, застосувань, ризиків та сучасних практик використання.

Дослідження підтвердило, що вказівники, які зберігають адреси пам'яті, є потужним інструментом C++, що забезпечує непряму адресацію, ефективну роботу з масивами через арифметику вказівників та можливість низькорівневого доступу. Було розглянуто ключові операції (&, \*, ->), типи вказівників (void\*, nullptr) та їх взаємодію з const.

Аналіз показав критичну роль вказівників у керуванні динамічною пам'яттю за допомогою new та delete. Водночас було наголошено на значних ризиках ручного керування: витоках пам'яті, висячих вказівниках та інших помилках, що загрожують стабільності та безпеці програм. Також було розглянуто застосування вказівників для реалізації динамічних структур даних, поліморфізму через віртуальні функції та використання вказівників на функції.

Особливу увагу приділено сучасним підходам C++ до керування ресурсами. Принцип RAII та використання розумних вказівників (std::unique\_ptr, std::shared\_ptr) були представлені як ефективні та безпечніші альтернативи "сирим" вказівникам, що значно знижують ймовірність помилок, пов'язаних з пам'яттю.

Практична частина роботи, реалізована у вигляді програми керування списком студентів (Додатки А, Б), продемонструвала використання стандартних контейнерів (std::vector), які інкапсулюють роботу з динамічною пам'яттю, що опосередковано базується на вказівниках.

Загальний висновок роботи полягає в тому, що вказівники залишаються важливою частиною C++, надаючи унікальні можливості контролю та продуктивності. Однак їх використання вимагає глибокого розуміння та обережності. Застосування сучасних ідіом C++, зокрема RAII та розумних вказівників, є ключовим для написання надійного та безпечного коду.

## ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Страуструп Б. The C++ Programming Language. 4-е видання. Addison-Wesley, 2013. — 1376 с.
2. Мейєрс С. Эффективное использование C++: 55 верных способов улучшить структуру и код программ. 3-е видання. — М.: Вильямс, 2005. — 352 с.
3. Sutter H., Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. — Addison-Wesley, 2005. — 240 с.
4. ISO/IEC 14882:2020. Programming languages — C++. Standard ISO/IEC, 2020. — 1800 с.
5. cppreference.com. Довідник зі стандарту мови програмування C++ [Електронний ресурс]. — Режим доступу: https://en.cppreference.com/ — Дата звернення: 27.04.2025.
6. Valgrind Documentation. Офіційна документація інструменту для перевірки пам'яті [Електронний ресурс]. — Режим доступу: https://valgrind.org/docs/ — Дата звернення: 27.04.2025.
7. Microsoft Docs: Smart Pointers (C++) [Електронний ресурс]. — Режим доступу: <https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp> — Дата звернення: 27.04.2025.

ДОДАТКИ

**Додаток А**

Лістинг - Код програми

#include <iostream>

#include <string>

#include <vector>

#include <limits>

#include <ios>

struct Student {

std::string name;

double grade;

};

double readDouble(const char\* prompt) {

double value;

while (true) {

std::cout << prompt;

std::cin >> value;

if (std::cin.good()) {

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

return value;

} else {

std::cout << "Некоректне введення. Будь ласка, введіть число.\n";

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

}

}

std::string readString(const char\* prompt) {

std::string value;

std::cout << prompt;

std::getline(std::cin >> std::ws, value);

return value;

}

void addStudent(std::vector<Student>& student\_list) {

Student new\_student;

new\_student.name = readString("Введіть ім'я студента: ");

new\_student.grade = readDouble("Введіть оцінку студента: ");

student\_list.push\_back(new\_student);

std::cout << "Студента додано.\n";

}

void showStudents(const std::vector<Student>& student\_list) {

if (student\_list.empty()) {

std::cout << "Список студентів порожній.\n";

} else {

std::cout << "\n--- Список студентів ---" << std::endl;

for (const Student& s : student\_list) {

std::cout << "Ім'я: " << s.name << ", Оцінка: " << s.grade << std::endl;

}

}

}

int main() {

std::vector<Student> student\_list;

int choice;

do {

std::cout << "\n--- Меню ---" << std::endl;

std::cout << "1. Додати студента та оцінку" << std::endl;

std::cout << "2. Показати список студентів" << std::endl;

std::cout << "0. Вихід" << std::endl;

choice = static\_cast<int>(readDouble("Ваш вибір: "));

switch (choice) {

case 1:

addStudent(student\_list);

break;

case 2:

showStudents(student\_list);

break;

case 0:

std::cout << "Вихід з програми.\n";

break;

default:

std::cout << "Невірний вибір. Спробуйте ще раз.\n";

break;

}

} while (choice != 0);

return 0;

}

**Додаток Б**

Рисунок - Блок-схема роботи програми

