УКРАЇНА

ЖИТОМИРСЬКИЙ КООПЕРАТИВНИЙ ФАХОВИЙ КОЛЕДЖ БІЗНЕСУ І ПРАВА

НАВЧАЛЬНА ПРАКТИКА З ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Методичні рекомендації для самостійної роботи студентів

Спеціальність: F2 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

Укладач: Олександр ФІЛІНСЬКИЙ – спеціаліст

Затверджено
на засіданні циклової комісії інженерії
програмного забезпечення та математики
Протокол № ___ від «____» _____ 2026 р.
Голова комісії _____ Наталія ГОНГАЛО

Житомир 2026 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

Методичні рекомендації для самостійної роботи студентів спеціальності F2 «Інженерія програмного забезпечення» складено відповідно до програми обов'язкової навчальної дисципліни «Навчальна практика з об'єктно-орієнтованого програмування», розробленої для підготовки фахових молодших бакалаврів за освітньо-професійною програмою «Інженерія програмного забезпечення».

Самостійна робота студента — це форма організації освітнього процесу, під час якої заплановані завдання виконуються студентом під методичним керівництвом викладача, але без його безпосередньої участі. Згідно з навчальним планом, на самостійне опрацювання дисципліни відведено 108 годин (загальний обсяг дисципліни — 180 годин, з них 72 години практичних занять).

Мета методичних рекомендацій — надати студентам допомогу під час опрацювання питань дисципліни, винесених на самостійне вивчення, та сприяти формуванню стійких навичок проєктування та реалізації програмних застосунків мовою C++ із використанням об'єктно-орієнтованого підходу, сучасних стандартів мови (C++17, C++20) та стандартної бібліотеки шаблонів (STL).

У методичних рекомендаціях за кожною з шістнадцяти тем дисципліни наведено перелік питань для самостійного вивчення, методичні рекомендації до їх опрацювання, практичне завдання для самостійної роботи студента (СРС) та запитання для самоперевірки. Наприкінці подано критерії оцінювання та список рекомендованої літератури й інформаційних ресурсів; номери джерел у розділі «Література» кожної теми відповідають нумерації цього списку.

Практичні завдання виконуються за стандартом C++20 у середовищі Visual Studio Code з компілятором GCC 14 (MSYS2) та системою збирання CMake або у Visual Studio 2022 Community; для швидкої перевірки фрагментів коду та запуску санітайзерів зручно використовувати Compiler

Explorer (https://godbolt.org). Кожне практичне завдання оформлюється як проєкт CMake за шаблоном ПР № 1 і має збиратися з параметрами `-Wall -Wextra` без попереджень. Усі завдання рекомендується вести в репозиторії Git.

Опрацьовувати питання самостійної роботи рекомендується в такій послідовності:

- ознайомитися з темою та питаннями для самостійного вивчення;
- вивчити питання, скориставшись методичними рекомендаціями та рекомендованою літературою;
- скласти стислий конспект опрацьованих питань;
- виконати практичне завдання, перевірити роботу програми на кількох наборах даних (за можливості — модульними тестами) і зафіксувати код у репозиторії Git;
- дати відповіді на запитання для самоперевірки.

Навчальний матеріал для самостійного вивчення виноситься на поточний і підсумковий контроль знань. За бажанням викладач може замінити або доповнити запитання для самоперевірки завданнями: підготувати реферат, презентацію, виконати тестування тощо.

Розподіл годин самостійної роботи за темами наведено в таблиці 1.

Таблиця 1

Розподіл годин самостійної роботи за темами

№	Назва теми	Кількість годин
1	Вступ до навчальної практики. Сучасне середовище розробки мовою C++	4
2	Класи та об’єкти. Інкапсуляція	6
3	Керування ресурсами. Копіювання та переміщення об’єктів	12
4	Перевантаження операцій. Функціональні об’єкти та лямбда-вирази	6
5	Успадкування	6
6	Поліморфізм. Абстрактні класи та інтерфейси	6
7	Розумні вказівники та сучасне керування пам’яттю	4
8	Опрацювання виключних ситуацій	6
9	Шаблони функцій і класів. Узагальнене програмування	12

10	Послідовні контейнери та ітератори STL	6
11	Асоціативні контейнери та адаптери контейнерів	8
12	Алгоритми STL. Бібліотека діапазонів	6
13	Рядки, потоки введення-виведення та файлова система	4
14	Багатопотокове програмування в C++	4
15	Принципи проєктування. Шаблони проєктування. Модульне тестування	4
16	Практикум з програмування: розроблення та захист прикладного проєкту	14
	Разом	**108**

ТЕМА 1. ВСТУП ДО НАВЧАЛЬНОЇ ПРАКТИКИ. СУЧАСНЕ СЕРЕДОВИЩЕ РОЗРОБКИ МОВОЮ C++

Література: *2, 8; інформаційні ресурси: 12, 19, 20, 21, 31.*

Кількість годин для самостійної роботи: *4.*

Питання для самостійного вивчення:

1. Еволюція стандартів C++: основні нововведення C++11, C++14, C++17, C++20 та C++23.
2. Встановлення компілятора GCC (MSYS2), CMake і Git на домашньому комп'ютері. Багатофайловий проєкт CMake, параметри -std=c++20 -Wall -Wextra.
3. Санітайзери (AddressSanitizer) як засіб пошуку помилок роботи з пам'яттю.
4. Git і GitHub: команди init, add, commit, branch, merge, push.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Розпочніть з огляду розвитку мови. Стандарт C++11 фактично створив «сучасний C++»: `auto`, цикл range-based for, лямбда-вирази, семантику переміщення, розумні вказівники. C++14 і C++17 розвинули ці ідеї (`std::make_unique`, `std::optional`, `std::variant`, структуровані прив'язки, `<filesystem>`), а C++20 додав концепти, оператор `<=>`, бібліотеку діапазонів `<ranges>` і `std::format`. Складіть таблицю «Стандарт — рік — три головні нововведення» для C++11–C++23; вона знадобиться протягом усього курсу, щоб розуміти, чому певна можливість не працює зі старим компілятором.

Встановіть інструменти на домашньому комп'ютері так само, як описано в ПР № 1: MSYS2 з пакетами GCC, CMake і Ninja, Visual Studio Code з розширеннями C/C++ та CMake Tools (або Visual Studio 2022 Community), Git. Перевірте версії командами `g++ --version`, `cmake --version`, `git --version`. Версія GCC має бути не нижче 13, інакше не працюватиме `std::format`.

Опануйте файл `CMakeLists.txt`: команда `project` задає ім'я проєкту, `set(CMAKE_CXX_STANDARD 20)` — стандарт мови, `add_compile_options(-Wall`

`-Wextra`) — попередження, `add_executable` — ім'я програми та список файлів `.cpp`. Створіть проєкт із трьох файлів (`main.cpp`, `utils.h`, `utils.cpp`), зберіть його і навмисно додайте кілька помилок, що породжують попередження (невикористана змінна, порівняння знакового й беззнакового, неініціалізована змінна). Сформулюйте для себе правило: програма вважається готовою лише тоді, коли збирається без попереджень.

Ознайомтеся із санітайзерами. AddressSanitizer — це режим компілятора (`-fsanitize=address`), у якому програма під час виконання перевіряє кожне звертання до пам'яті та повідомляє про вихід за межі масиву, використання звільненої пам'яті, витоки. У MSYS2 під Windows санітайзери не працюють, тому виконуйте ці експерименти в Compiler Explorer (godbolt.org) або на Linux.

Відпрацюйте роботу з Git: створення репозиторію (`git init`), фіксацію змін (`git add`, `git commit`), створення гілки для нової функції (`git branch feature`, `git switch feature`), злиття (`git merge`) та завантаження на GitHub (`git remote add origin …`, `git push`). Ведіть у Git усі наступні практичні роботи — у кінці курсу історія комітів стане частиною захисту проєкту.

ПРАКТИЧНЕ ЗАВДАННЯ

Створити на GitHub репозиторій `oop-practice` і завантажити до нього проєкт CMake з трьох файлів: структура `Student`, функції введення, сортування за середнім балом і виведення списку студентів. Проєкт має збиратися з параметрами `-std=c++20 -Wall -Wextra` без попереджень. Розробити нову функцію (пошук студента за прізвищем) в окремій гілці та злити її з основною. Перевірити в Compiler Explorer із `-fsanitize=address` фрагмент з виходом за межі масиву та записати в конспект повідомлення санітайзера. Підготувати таблицю нововведень стандартів C++11–C++23 (по три для кожного).

Запитання для самоперевірки:

1. Які основні можливості з'явилися в стандартах C++11 і C++20?

2. Які команди містить мінімальний файл `CMakeLists.txt` і для чого кожна з них?
3. Для чого в Git використовують гілки? Як злити гілку з основною?

ТЕМА 2. КЛАСИ ТА ОБ'ЄКТИ. ІНКАПСУЛЯЦІЯ

Література: *1, 2, 3, 8; інформаційні ресурси: 9, 14, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Перетворення структури на клас: закриті поля, інваріанти, методи доступу. Const-коректність.
2. Види конструкторів: за замовчуванням, параметризований, делегувальний, explicit. Список ініціалізації, = default, = delete.
3. Порядок створення та знищення об'єктів. Статичні поля і методи, покажчик this, ланцюжки викликів.
4. Розділення класу на файли .h та .cpp.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Головна відмінність класу від структури — не синтаксис, а **відповідальність**: клас гарантує, що його об'єкт завжди перебуває у правильному стані. Ця умова називається **інваріантом** (наприклад, «знаменник дробу не дорівнює нулю», «баланс рахунку не від'ємний»). Щоб інваріант не можна було порушити ззовні, поля роблять `private`, а будь-яка зміна відбувається лише через методи, які перевіряють нове значення. Візьміть структуру «Товар» або «Студент» з попереднього курсу, перетворіть її на клас і сформулюйте письмово інваріанти кожного поля.

Опрацюйте конструктори. Поля ініціалізуйте у **списку ініціалізації** (`Point(double x, double y) : x_(x), y_(y) {}`), а не присвоєнням у тілі: так константні поля й посилання взагалі можна ініціалізувати, а інші — без зайвої роботи. Пам'ятайте, що поля ініціалізуються в порядку їх оголошення в класі, а не в порядку запису у списку (компілятор попередить про це параметром `-Wreorder`). **Делегувальний** конструктор викликає інший

конструктор того самого класу, що усуває дублювання коду. Конструктор з одним параметром позначайте `explicit`, щоб заборонити неочікувані неявні перетворення. Конструкції `= default` і `= delete` дозволяють явно залишити або заборонити згенеровану компілятором функцію.

Простежте життєвий цикл об'єктів: додайте до конструкторів і деструктора виведення повідомлень і створіть кілька об'єктів у різних областях видимості. Ви побачите, що локальні об'єкти знищуються у зворотному порядку створення. Реалізуйте лічильник «живих» об'єктів на **статичному полі** — воно спільне для всіх об'єктів класу. Метод, що повертає `*this` за посиланням, дозволяє будувати ланцюжки викликів: `builder.setWidth(10).setHeight(5);`.

Засвойте const-коректність: методи, що лише читають стан (`getName`, `area`, `print`), позначайте словом `const`. Лише такі методи можна викликати для константного об'єкта або через `const &`. Великі класи розділяйте на заголовковий файл (оголошення класу) і файл реалізації (визначення методів з операцією `::`).

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити клас `Calendar` (поля: день, місяць, рік; інваріант — дата коректна з урахуванням високосних років) із параметризованим, делегувальним (лише рік — 1 січня) та `explicit`-конструктором з рядка формату `"DD.MM.YYYY"`, методами `nextDay()` та `addDays(int)`, що повертають `*this` для ланцюжків, константними методами `toString()` і `dayOfYear()`, статичним лічильником створених об'єктів. Розмістити клас у файлах `calendar.h` і `calendar.cpp`. У `main()` продемонструвати всі конструктори, спробу створити некоректну дату (виключення `std::invalid_argument`), ланцюжок `d.nextDay().addDays(30)` і порядок знищення об'єктів.

Запитання для самоперевірки:

1. Що таке інваріант класу і як інкапсуляція допомагає його зберегти?

2. Чому поля краще ініціалізувати у списку ініціалізації? У якому порядку вони ініціалізуються?
3. Для чого використовують `explicit`, `= default` і `= delete`?

ТЕМА 3. КЕРУВАННЯ РЕСУРСАМИ. КОПІЮВАННЯ ТА ПЕРЕМІЩЕННЯ ОБ'ЄКТІВ

Література: *1, 2, 8; інформаційні ресурси: 9, 10, 12, 14, 23.*

Кількість годин для самостійної роботи: *12.*

Питання для самостійного вивчення:

1. Ідіома RAII. Клас-обгортка над динамічним масивом (власний аналог std::vector).
2. Поверхневе та глибоке копіювання. Конструктор копіювання й оператор присвоєння. Ідіома copy-and-swap.
3. Семантика переміщення: rvalue-посилання, конструктор і оператор присвоєння переміщенням, noexcept, std::move.
4. Правила трьох, п'яти та нуля. Класи з забороненим копіюванням. Перевірка витоків пам'яті.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

RAII (Resource Acquisition Is Initialization — «отримання ресурсу є ініціалізацією») — центральна ідея сучасного C++: ресурс (пам'ять, файл, м'ютекс) захоплюється в конструкторі об'єкта і гарантовано звільняється в деструкторі. Оскільки деструктор локального об'єкта викликається автоматично під час виходу з блоку — навіть у разі виключення, — ресурс не може «загубитися». Розробіть клас `DynArray` з полями `int* data_` і `std::size_t size_`, що виділяє пам'ять у конструкторі (`new int[n]`) і звільняє в деструкторі (`delete[]`).

Тепер скопіюйте такий об'єкт: `DynArray b = a;`. Згенерований компілятором конструктор копіювання скопіює лише покажчик (**поверхневе копіювання**), обидва об'єкти вказуватимуть на ту саму пам'ять, і під час знищення вона буде звільнена двічі. Запустіть цей код із санітайзером і

запишіть повідомлення `double-free`. Виправлення — **глибоке копіювання**: власний конструктор копіювання виділяє нову пам'ять і копіює елементи. Оператор присвоєння копіюванням зручно реалізувати ідіомою **copy-and-swap**: параметр приймається за значенням (копія), а потім вміст обмінюється з `*this` через `std::swap`; це автоматично безпечно щодо самоприсвоєння та виключень.

Опрацюйте **переміщення**. Тимчасовий об'єкт (rvalue) все одно буде знищено, тому замість копіювання його ресурс можна просто «забрати»: конструктор переміщення `DynArray(DynArray&& other) noexcept` переносить покажчик і обнуляє його в `other`. Функція `std::move` нічого не переміщує — вона лише перетворює вираз на rvalue, дозволяючи вибрати перемістову версію. Позначайте переміщення `noexcept`: інакше `std::vector` під час розширення копіюватиме ваші об'єкти замість переміщення. Виміряйте час копіювання та переміщення вектора з мільйона елементів і порівняйте.

Узагальніть правила. **Правило трьох**: якщо класу потрібен власний деструктор, конструктор копіювання або оператор присвоєння копіюванням — найімовірніше, потрібні всі три. **Правило п'яти** додає до них дві перемістові операції. **Правило нуля**: найкраще взагалі не писати жодної з цих функцій, а зберігати ресурси в готових RAII-класах (`std::vector`, `std::string`, `std::unique_ptr`), тоді компілятор згенерує правильні операції сам. Для класів, які не можна копіювати (наприклад, «з'єднання з файлом»), явно забороніть копіювання: `NoCopy(const NoCopy&) = delete;`.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити клас `IntVector` (власний аналог `std::vector<int>`) з ідіомою RAII: конструктори (порожній, з розміром, з `std::initializer_list<int>`), деструктор, повний набір правила п'яти (копіювання глибоке, присвоєння через copy-and-swap, переміщення з `noexcept`), методи `push_back` (з подвоєнням ємності), `size`, `capacity`, `operator[]`, `at` (з перевіркою меж). Додати статичні лічильники копіювань і

переміщень. У `main()` продемонструвати кожну операцію, вивести значення лічильників для `std::vector<IntVector>` із 5 елементів з `noexcept` і без нього та перевірити програму санітайзером на відсутність витоків. Окремо створити клас `LogFile` із забороненим копіюванням, але дозволеним переміщенням.

Запитання для самоперевірки:

1. Що таке RAII і чому ця ідіома гарантує звільнення ресурсу навіть у разі виключення?
2. Чим поверхневе копіювання відрізняється від глибокого? До якої помилки призводить поверхневе копіювання класу з динамічною пам’яттю?
3. Сформулюйте правила трьох, п’яти та нуля. Чому перемістові операції слід позначати `noexcept`?

ТЕМА 4. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ. ФУНКЦІОНАЛЬНІ ОБ’ЄКТИ ТА ЛЯМБДА-ВИРАЗИ

Література: *1, 2, 3, 8; інформаційні ресурси: 9, 14, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Перевантаження арифметичних операцій, операцій порівняння, індексування [], виклику функції () та потокових операцій << і >>.
2. Оператор <=> та автоматично згенеровані операції порівняння (C++20).
3. Конвертувальні конструктори та оператори перетворення типу з explicit.
4. Функціональні об’єкти. Лямбда-вирази: способи захоплення, узагальнені лямбди, std::function.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Перевантаження операцій робить власні типи такими ж зручними, як вбудовані: `Complex c = a + b * 2;`. Дотримуйтеся правила: операції, що змінюють об’єкт (`+=`, `[]`, `++`, `=`), оголошуйте **методами класу**, а симетричні

бінарні операції (+, *, ==) та потокові (<<, >>) — **вільними функціями** (за потреби дружніми). Ефективний прийом: спочатку реалізувати operator+=, а operator+ записати через нього. Оператор [] перевантажуйте у двох версіях — звичайній і const.

У C++20 достатньо оголосити auto operator<=>(const T&) const = default;, і компілятор сам згенерує всі шість операцій порівняння, порівнюючи поля в порядку оголошення. Якщо порядок має бути інший (наприклад, дроби порівнюються за значенням, а не за чисельником), реалізуйте <=> вручну, повертаючи std::strong_ordering або std::weak_ordering, і окремо operator==. Розгляньте операції інкременту: префіксна версія T& operator++() повертає посилання, постфіксна T operator++(int) — копію попереднього значення.

Користувацькі перетворення типів бувають двох видів: конвертувальний конструктор (Fraction(int n) — з int у дріб) та оператор перетворення (explicit operator double() const — з дробу в double). Позначайте їх explicit, якщо неявне перетворення може здивувати користувача класу. Загалом дотримуйтеся **принципу найменшого здивування**: операція + не повинна змінювати операнди, а == — бути несиметричною.

Функціональний об’єкт (функтор) — це об’єкт класу з перевантаженою операцією (); на відміну від звичайної функції, він може зберігати стан. Лямбда-вираз — короткий запис такого об’єкта: [захоплення](параметри) { тіло }. Розберіть способи захоплення: [x] — копія, [&x] — посилання, [=] і [&] — усі змінні, [this] — поточний об’єкт. Узагальнена лямбда приймає параметри типу auto. Щоб зберегти лямбду в змінній або векторі з довільним типом, використовуйте std::function<int(int)>.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити клас Matrix (розмір задається в конструкторі, елементи double зберігаються в std::vector<double>) з операціями +, -, * (матриць і

матриці на число), `+=`, `==`, `operator()(i, j)` для доступу до елемента, потоковими `<<` і `>>`, з перевіркою узгодженості розмірів (виключення). Окремо розробити клас `Fraction` з автоматичним скороченням, операціями `+ - * /`, оператором `<=>` (порівняння за значенням), префіксним і постфіксним `++` та `explicit operator double`. Написати програму «Калькулятор операцій», де операції `+ - * /` зберігаються в `std::map<char, std::function<Fraction(Fraction, Fraction)>>` у вигляді лямбда-виразів.

Запитання для самоперевірки:

1. Які операції краще перевантажувати як методи класу, а які — як вільні функції? Чому?
2. Що генерує компілятор для `operator<=>` з `= default`?
3. Які способи захоплення змінних існують у лямбда-виразах? Чим відрізняються `[x]` і `[&x]`?

ТЕМА 5. УСПАДКУВАННЯ

Література: *1, 2, 3, 8; інформаційні ресурси: 14, 18, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Діаграми класів UML. Одиночне та багаторівневе успадкування. Порядок виклику конструкторів і деструкторів.
2. Передавання параметрів у конструктор базового класу. Успадкування конструкторів (using).
3. Множинне успадкування, проблема ромба та віртуальне успадкування.
4. Зрізання об’єктів (slicing). Порівняння успадкування та композиції.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Перш ніж писати ієрархію, намалюйте **діаграму класів UML**: кожен клас — прямокутник з трьох частин (ім’я, поля, методи), видимість позначають `+` (public), `-` (private), `#` (protected), успадкування — суцільна лінія з порожнім трикутником, спрямованим до базового класу. Діаграма допомагає перевірити ієрархію простим тестом: похідний клас має

відповідати на питання «**є** різновидом базового?» («Автобус є транспортом» — так; «Двигун є автомобілем» — ні).

Під час створення об’єкта похідного класу спочатку повністю конструюється базова частина, потім поля похідного класу і лише потім виконується тіло його конструктора; деструктори викликаються у зворотному порядку. Параметри конструктору базового класу передають у списку ініціалізації: `Bus(std::string model, int seats) : Vehicle(std::move(model)), seats_(seats) {}`. Якщо похідний клас не додає нових полів, конструктори базового можна успадкувати: `using Vehicle::Vehicle;`.

Змоделюйте **проблему ромба**: класи `Scanner` і `Printer` успадковують `Device`, а `Mfp` (багатофункціональний пристрій) — обидва. Об’єкт `Mfp` міститиме дві копії `Device`, і звертання до її полів стане неоднозначним (`request for member is ambiguous`). Віртуальне успадкування (`class Scanner : virtual public Device`) залишає лише одну спільну копію, але тоді конструктор `Device` викликає безпосередньо найпохідніший клас. Застосовуйте множинне успадкування обережно, переважно для інтерфейсів.

Дослідіть **зрізання**: якщо присвоїти об’єкт похідного класу змінній базового типу за значенням (`Vehicle v = bus;`), поля похідного класу буде «відрізано». Тому поліморфні об’єкти передають за посиланням або покажчиком. Нарешті, порівняйте успадкування з **композицією** (клас містить об’єкт іншого класу як поле, відношення «має»). Реалізуйте одну задачу обома способами й зробіть висновок; загальне правило — віддавати перевагу композиції, а успадкування використовувати лише для справжнього відношення «є».

ПРАКТИЧНЕ ЗАВДАННЯ

Побудувати діаграму класів і реалізувати ієрархію «Транспорт»: базовий клас `Vehicle` (модель, рік, максимальна швидкість), похідні `Car` (кількість дверей), `Bus` (кількість місць), `ElectricBus` (успадковує `Bus`, ємність батареї) — з виведенням повідомлень у конструкторах і деструкторах

для спостереження за їх порядком. Застосувати успадкування конструкторів, перевизначити метод `info()`, викликати версію базового класу через `Vehicle::info()`. Продемонструвати зрізання об'єкта. Змоделювати ромб `Device` → `Scanner`, `Printer` → `Mfp` без віртуального та з віртуальним успадкуванням і порівняти результати. Реалізувати клас `Car` з двигуном через композицію (клас `Engine` як поле).

Запитання для самоперевірки:

1. У якому порядку викликаються конструктори та деструктори в багаторівневій ієрархії?
2. У чому полягає проблема ромба і як її розв'язує віртуальне успадкування?
3. Що таке зрізання об'єктів? Коли варто обрати композицію замість успадкування?

ТЕМА 6. ПОЛІМОРФІЗМ. АБСТРАКТНІ КЛАСИ ТА ІНТЕРФЕЙСИ

Література: *1, 2, 3, 8; інформаційні ресурси: 9, 10, 14, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Віртуальні методи, специфікатори override та final. Таблиця віртуальних методів.
2. Абстрактні класи та інтерфейси. Зберігання поліморфних об'єктів у std::vector.
3. Віртуальний деструктор. Виклик віртуальних функцій у конструкторах і деструкторах.
4. Віртуальне копіювання (clone). dynamic_cast, typeid, static_cast. Статичний і динамічний поліморфізм.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Динамічний поліморфізм — це здатність викликати через покажчик або посилання на базовий клас метод того класу, яким об'єкт є насправді. Для цього метод оголошують `virtual`, а в похідних класах перевизначають з

обов’язковим специфікатором `override` — тоді компілятор повідомить про помилку, якщо сигнатура не збігається. Специфікатор `final` забороняє подальше перевизначення методу або успадкування класу. Механізм реалізується через **таблицю віртуальних методів** (vtable): кожен поліморфний об’єкт містить прихований покажчик на таблицю адрес методів свого класу. Відкрийте свій код у C++ Insights (cppinsights.io), щоб побачити, що генерує компілятор.

Абстрактний клас містить хоча б один чисто віртуальний метод (`virtual double area() const = 0;`) і не може мати об’єктів. Клас, що складається лише з чисто віртуальних методів і віртуального деструктора, відіграє роль **інтерфейсу** (`IPrintable`, `ISerializable`). Поліморфні об’єкти зберігайте у векторі покажчиків на базовий клас (у наступній темі — розумних вказівників) і обробляйте одним циклом: `for (const auto& s : shapes) total += s->area();`.

Обов’язково дослідіть дві пастки. Перша — **невіртуальний деструктор**: якщо видаляти об’єкт похідного класу через покажчик на базовий, деструктор похідного класу не викличеться, і його ресурси не звільняться. Правило: базовий клас поліморфної ієрархії завжди має `virtual ~Base() = default;`. Друга — виклик віртуального методу з конструктора або деструктора базового класу: у цей момент об’єкт «ще не» або «вже не» є похідним, тому буде викликана версія базового класу.

Скопіювати об’єкт, маючи лише покажчик на базовий клас, звичайним конструктором копіювання неможливо — тип невідомий. Рішення — **віртуальне копіювання**: метод `virtual std::unique_ptr<Shape> clone() const`, який кожен похідний клас реалізує через власний конструктор копіювання. `dynamic_cast` безпечно перетворює покажчик на похідний тип і повертає `nullptr`, якщо об’єкт іншого типу; `typeid` повідомляє фактичний тип. Часте використання `dynamic_cast` зазвичай свідчить про невдалий дизайн — краще додати віртуальний метод. Порівняйте динамічний

поліморфізм зі статичним (перевантаження функцій, шаблони), де вибір відбувається під час компіляції.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити поліморфну ієрархію «Банківські рахунки»: інтерфейс `IPrintable` (метод `print`), абстрактний клас `Account` (номер, власник, баланс; чисто віртуальні `monthlyUpdate()` і `clone()`; віртуальний деструктор), похідні `SavingsAccount` (нарахування відсотків), `CreditAccount` (ліміт і комісія), `DepositAccount final` (заборона зняття до кінця терміну). Зберегти рахунки у `std::vector` покажчиків на `Account`, застосувати до всіх `monthlyUpdate()`, вивести загальний баланс, створити повну копію вектора через `clone()`, знайти всі кредитні рахунки за допомогою `dynamic_cast`. Окремо продемонструвати проблему невіртуального деструктора (без і з `virtual`) і виклик віртуального методу з конструктора.

Запитання для самоперевірки:

1. Для чого потрібні специфікатори `override` і `final`?
2. Чим абстрактний клас відрізняється від інтерфейсу? Чому не можна створити об’єкт абстрактного класу?
3. Що станеться, якщо базовий клас поліморфної ієрархії не має віртуального деструктора?

ТЕМА 7. РОЗУМНІ ВКАЗІВНИКИ ТА СУЧАСНЕ КЕРУВАННЯ ПАМ’ЯТТЮ

Література: *2, 3, 8; інформаційні ресурси: 9, 12, 14, 23.*

Кількість годин для самостійної роботи: *4.*

Питання для самостійного вивчення:

1. Типові помилки роботи з «сирими» покажчиками: витік пам’яті, висячий покажчик, подвійне видалення.
2. std::unique_ptr і std::shared_ptr. Передавання власності через std::move. std::make_unique, std::make_shared.

3. std::weak_ptr і розрив циклічних залежностей. Лічильник посилань use_count.
4. Користувацька функція видалення. Вибір типу вказівника для параметрів і значень, що повертаються.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Розпочніть з аналізу помилок «сирих» покажчиків. **Витік** — пам'ять виділено `new`, але не звільнено (наприклад, функція завершилася виключенням до `delete`). **Висячий покажчик** — пам'ять уже звільнено, а покажчик продовжують використовувати. **Подвійне видалення** — `delete` викликано двічі для тієї самої адреси. Усі три помилки мають спільну причину: незрозуміло, хто **володіє** об'єктом і відповідає за його знищення. Розумні вказівники роблять володіння явним.

`std::unique_ptr<T>` — **одноосібне володіння**: об'єкт знищується разом з вказівником; копіювати `unique_ptr` не можна, лише переміщувати (`auto b = std::move(a);` — після цього `a` порожній). Це вибір за замовчуванням. `std::shared_ptr<T>` — **спільне володіння**: кілька вказівників ділять один об'єкт, а лічильник посилань (`use_count()`) знищує його, коли зникає останній власник. Створюйте вказівники функціями `std::make_unique<T>(...)` і `std::make_shared<T>(...)` — без явного `new`.

Головна пастка `shared_ptr` — **циклічні посилання**: якщо два об'єкти тримають `shared_ptr` один на одного (батько ↔ дитина, сусідні вузли двозв'язного списку), лічильники ніколи не стануть нулем і пам'ять «витече». Розірвіть цикл: зворотне посилання зробіть `std::weak_ptr` — він спостерігає за об'єктом, не збільшуючи лічильник. Щоб скористатися об'єктом, викличте `lock()` і перевірте результат. Намалюйте граф володіння для двозв'язного списку і перевірте відсутність витоку санітайзером.

Засвойте настанови вибору (C++ Core Guidelines): функція, яка лише **використовує** об'єкт, приймає `T&` або `const T&` (або `T*`, якщо об'єкта може не бути); функція, яка **забирає** власність, приймає `std::unique_ptr<T>` за значенням; фабрика повертає `std::unique_ptr<T>`. Користувацька функція

видалення дозволяє керувати не лише пам’яттю: `std::unique_ptr<FILE, decltype(&fclose)> f(fopen("a.txt", "r"), &fclose);` автоматично закриє файл.

ПРАКТИЧНЕ ЗАВДАННЯ

Переробити ієрархію рахунків (або фігур) з теми 6, замінивши всі «сирі» покажчики на `std::unique_ptr` (вектор `std::vector<std::unique_ptr<Account>>`, фабрична функція `createAccount(type, …)`, метод `clone()` повертає `unique_ptr`), і переконатися санітайзером, що витоків немає. Реалізувати двозв’язний список `DList` із вузлами `next` — `std::shared_ptr`, `prev` — `std::weak_ptr`; вивести `use_count` вузлів, показати витік у версії з двома `shared_ptr` та його усунення. Створити `unique_ptr` з користувацькою функцією видалення для файлу `FILE*`, яка виводить повідомлення про закриття.

Запитання для самоперевірки:

1. Чим `std::unique_ptr` відрізняється від `std::shared_ptr`? Який з них обирати за замовчуванням?
2. Як виникає циклічне посилання між `shared_ptr` і як його розірвати?
3. Чому для створення розумних вказівників рекомендують `std::make_unique` і `std::make_shared`?

ТЕМА 8. ОПРАЦЮВАННЯ ВИКЛЮЧНИХ СИТУАЦІЙ

Література: *1, 2, 3, 8; інформаційні ресурси: 9, 14, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Коди повернення та виключення: порівняння підходів. Стандартні виключення (out_of_range, invalid_argument, bad_alloc).
2. Власна ієрархія класів виключень. Розкручування стеку та виклик деструкторів.
3. Гарантії безпеки щодо виключень (базова, строга, nothrow). Специфікатор noexcept.

4. std::optional, std::variant, std::visit як альтернатива виключенням. assert і static_assert.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Порівняйте два способи повідомити про помилку. **Код повернення** (функція повертає `-1` або `false`) легко проігнорувати, і він «займає» значення, що повертається. **Виключення** неможливо випадково проігнорувати: якщо його не перехоплено, програма завершиться. Виключення доречні для справді виняткових ситуацій (файл не відкрився, дані пошкоджено, порушено інваріант), а не для звичайного розгалуження логіки. Реалізуйте одну задачу (наприклад, ділення дробів або читання числа з рядка) обома способами й оцініть читабельність.

Вивчіть ієрархію стандартних виключень: усі вони походять від `std::exception` з віртуальним методом `what()`; `std::logic_error` (помилки програміста: `invalid_argument`, `out_of_range`) і `std::runtime_error` (помилки середовища). Перехоплюйте виключення **за константним посиланням** (`catch (const std::exception& e)`), від конкретніших типів до загальніших. Власні виключення успадковуйте від стандартних (`class InsufficientFunds : public std::runtime_error`) і додавайте корисні поля (сума, номер рахунку).

Під час генерування виключення відбувається **розкручування стеку**: функції завершуються одна за одною, і для всіх локальних об’єктів викликаються деструктори. Саме тому RAII робить код безпечним щодо виключень. Розрізняйте гарантії: **базова** — після виключення об’єкт коректний і ресурси не втрачено; **строга** — операція або виконується повністю, або стан не змінюється (досягається через «зроби копію, зміни копію, обміняй»); **nothrow** — операція ніколи не генерує виключення (позначається `noexcept`; обов’язково для деструкторів і переміщення).

Для очікуваної відсутності результату зручніше `std::optional<T>` (наприклад, пошук, який може нічого не знайти): `if (auto r = find(id)) use(*r);`. `std::variant<A, B>` зберігає значення одного з кількох типів

(наприклад, результат або повідомлення про помилку), а `std::visit` обробляє кожен варіант. `assert(умова)` перевіряє припущення програміста лише в режимі налагодження (вимикається в Release через `NDEBUG`), а `static_assert` перевіряє умову ще під час компіляції.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити класи «Банк» і «Рахунок» з власною ієрархією виключень `BankError` → `InsufficientFunds` (з полями «запитана сума» і «доступно»), `AccountNotFound`, `InvalidAmount`. Реалізувати переказ між рахунками зі **строгою** гарантією безпеки (у разі помилки баланси обох рахунків не змінюються) та перевірити це тестом. Пошук рахунку повертати як `std::optional`, результат розбору команди користувача (`"deposit 100"`, `"withdraw abc"`) — як `std::variant<Command, ParseError>` з обробкою через `std::visit`. Додати клас `Tracer`, що виводить повідомлення в конструкторі й деструкторі, для спостереження за розкручуванням стеку, та `static_assert`, що перевіряє розмір типу суми.

Запитання для самоперевірки:

1. Чому виключення рекомендують перехоплювати за константним посиланням?
2. Чим строга гарантія безпеки щодо виключень відрізняється від базової? Як досягти строгої гарантії?
3. У яких випадках `std::optional` зручніший за генерування виключення?

ТЕМА 9. ШАБЛОНИ ФУНКЦІЙ І КЛАСІВ. УЗАГАЛЬНЕНЕ ПРОГРАМУВАННЯ

Література: *1, 2, 8; інформаційні ресурси: 10, 12, 14, 23.*

Кількість годин для самостійної роботи: *12.*

Питання для самостійного вивчення:

1. Шаблони функцій: явне та неявне виведення типів, перевантаження шаблонів.

2. Шаблони класів (стек, черга, зв’язний список, матриця) з нетиповими параметрами. Повна та часткова спеціалізація.
3. Шаблони зі змінною кількістю параметрів, згортальні вирази. auto, decltype, CTAD. constexpr.
4. Концепти та requires (C++20). Бібліотека <type_traits> і static_assert. Повідомлення компілятора про помилки в шаблонах.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Шаблон — це «заготовка», з якої компілятор генерує функцію або клас для конкретного типу. Запис `template <typename T> T maxOf(T a, T b)` описує сімейство функцій; під час виклику `maxOf(3, 7)` тип `T = int` виводиться автоматично, а у випадку `maxOf(3, 7.5)` виведення неможливе (різні типи), і тип потрібно вказати явно: `maxOf<double>(3, 7.5)`. Важливо: код шаблону повністю розміщують у **заголовковому файлі**, бо компілятор має бачити його тіло в кожному місці використання; інакше виникне помилка компонувальника `undefined reference`.

Розробіть шаблони класів «Стек» і «Черга» на основі `std::vector` або власного зв’язного списку з вузлами `std::unique_ptr`. **Нетиповий параметр** шаблону — це значення, відоме під час компіляції: `template <typename T, std::size_t Rows, std::size_t Cols> class Matrix` — розмір матриці стає частиною типу, і додати матриці різних розмірів неможливо ще на етапі компіляції. **Спеціалізація** дозволяє задати окрему реалізацію для певного типу: повна (`template <> class Stack<bool>`) або часткова (`template <typename T> class Stack<T*>`).

Шаблон зі змінною кількістю параметрів (`template <typename... Args>`) приймає будь-яку кількість аргументів різних типів; **згортальний вираз** обробляє їх в одному рядку: `return (args + ...);` або `(std::cout << ... << args);`. Ключові слова `auto` і `decltype` допомагають працювати з типами, які важко записати, а CTAD (C++17) дозволяє не вказувати аргументи шаблону класу: `std::pair p{1, 2.5};`. Функції `constexpr` можуть обчислюватися ще під час компіляції.

Найважливіше нововведення C++20 у цій темі — **концепти**. Концепт формулює вимоги до типу: `template <typename T> concept Numeric = std::is_arithmetic_v<T>;`, а шаблон обмежують так: `template <Numeric T> T sum(...)` або за допомогою `requires`. Порівняйте повідомлення компілятора про помилку для шаблону без концепту (десятки рядків із глибини реалізації) та з концептом (коротке «constraints not satisfied» з назвою порушеної вимоги). Для перевірки властивостей типів використовуйте `<type_traits>` (`std::is_integral_v`, `std::is_same_v`) разом зі `static_assert`.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити бібліотеку шаблонів у заголовкових файлах: функції `maxOf`, `swapValues`, `bubbleSort` (з довільним компаратором), `printAll(args...)` на згортальному виразі, `sumAll` з обмеженням концептом `Numeric`; шаблони класів `Stack<T>`, `Queue<T>` на власному зв'язному списку з `std::unique_ptr`, `Matrix<T, Rows, Cols>` з операціями `+` і `*` (множення можливе лише за узгоджених розмірів — перевірка на етапі компіляції), спеціалізацію `Stack<bool>` або `printValue<const char*>`. Продемонструвати роботу шаблонів для `int`, `double`, `std::string` і власного класу; навмисно викликати `sumAll` для рядків і записати в конспект повідомлення компілятора з концептом і без нього.

Запитання для самоперевірки:

1. Чому код шаблонів розміщують у заголовкових файлах?
2. Що таке нетиповий параметр шаблону? Наведіть приклад.
3. Яку перевагу дають концепти C++20 порівняно з необмеженими шаблонами?

ТЕМА 10. ПОСЛІДОВНІ КОНТЕЙНЕРИ ТА ІТЕРАТОРИ STL

Література: *2, 8; інформаційні ресурси: 13, 14, 18, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Складність основних операцій контейнерів std::array, std::vector, std::deque, std::list, std::forward_list. Вибір контейнера.
2. Ємність std::vector: size, capacity, reserve, shrink_to_fit. Експериментальне порівняння швидкодії контейнерів.
3. Категорії ітераторів. Зворотні ітератори. Інвалідація ітераторів. Ідіома erase–remove.
4. std::pair, std::tuple, структуровані прив'язки, std::span.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Складіть таблицю складності операцій: доступ за індексом, вставлення/видалення на початку, в кінці та в середині для п'яти послідовних контейнерів. `std::vector` зберігає елементи неперервно, тому доступ за індексом і додавання в кінець дуже швидкі, а вставлення в середину вимагає зсуву. `std::deque` швидко додає з обох кінців. `std::list` (двозв'язний) і `std::forward_list` (однозв'язний) вставляють за O(1) у будь-якому місці, але не мають доступу за індексом. `std::array` має фіксований розмір. На практиці **`std::vector` — вибір за замовчуванням**: завдяки кешу процесора він часто виявляється швидшим за `list` навіть там, де теорія обіцяє протилежне. Перевірте це експериментом на 100 000 елементів, вимірюючи час за допомогою `<chrono>`.

Розберіться з ємністю вектора: `size()` — кількість елементів, `capacity()` — скільки поміститься без перерозподілу пам'яті. Коли місце закінчується, вектор виділяє більший блок (у GCC — удвічі, у MSVC — у 1,5 раза) і переносить елементи. Якщо кількість елементів відома заздалегідь, викличте `reserve(n)` — це усуне зайві перерозподіли.

Ітератор — це узагальнений покажчик на елемент контейнера: `begin()` вказує на перший елемент, `end()` — на позицію **після** останнього; `rbegin()`/`rend()` дозволяють обхід у зворотному порядку. **Інвалідація** — ситуація, коли після зміни контейнера ітератор стає недійсним: для вектора будь-яке вставлення з перерозподілом робить недійсними всі ітератори, а видалення — ітератори після місця видалення. Типова помилка — видаляти елементи в циклі `for` з ітератором. Правильно: використовувати значення, яке повертає `erase`, або ідіому **erase–remove**: `v.erase(std::remove_if(v.begin(), v.end(), pred), v.end());` (у C++20 — коротше: `std::erase_if(v, pred);`).

Для зберігання кількох значень разом використовуйте `std::pair` і `std::tuple`, а розпаковуйте їх **структурованими прив'язками**: `auto [name, score] = results[0];`. `std::span<int>` (C++20) — «вікно» на неперервну послідовність (масив, вектор, `std::array`) без копіювання; це зручний тип параметра функції, що обробляє масив будь-якого походження.

ПРАКТИЧНЕ ЗАВДАННЯ

Провести експериментальне порівняння `std::vector`, `std::deque` і `std::list`: для 10 000, 50 000 і 100 000 елементів виміряти час вставлення в кінець, на початок і в середину, видалення всіх парних чисел та обчислення суми; результати оформити таблицею з висновками. Написати програму, що виводить зміни `size()` і `capacity()` під час додавання 20 елементів у вектор з `reserve` і без нього. Реалізувати функцію видалення елементів за умовою трьома способами (неправильний цикл з `erase`, правильний цикл, ідіома erase–remove) та пояснити помилку першого. Написати функцію `stats(std::span<const double>)`, що повертає `std::tuple` (мінімум, максимум, середнє), і викликати її для масиву, `std::array` та `std::vector` зі структурованими прив'язками.

Запитання для самоперевірки:

1. Чому `std::vector` вважають контейнером за замовчуванням? Коли доцільно обрати `std::list`?
2. Що таке інвалідація ітераторів? Як правильно видаляти елементи вектора за умовою?
3. Чим відрізняються `size()` і `capacity()`? Для чого використовують `reserve()`?

ТЕМА 11. АСОЦІАТИВНІ КОНТЕЙНЕРИ ТА АДАПТЕРИ КОНТЕЙНЕРІВ

Література: *2, 8; інформаційні ресурси: 14, 18, 22, 24.*

Кількість годин для самостійної роботи: *8.*

Питання для самостійного вивчення:

1. Контейнери std::set, std::multiset, std::map, std::multimap. Користувацькі компаратори.
2. Контейнери std::unordered_set, std::unordered_map. Хеш-функція для власного класу. Порівняння швидкодії.
3. Відмінність operator[] та at. Методи try_emplace, contains, lower_bound.
4. Адаптери std::stack, std::queue, std::priority_queue та задачі на їх використання.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Асоціативні контейнери зберігають елементи не за позицією, а за **ключем**. `std::set` — множина унікальних ключів, `std::map` — пари «ключ — значення»; обидва побудовані на збалансованому дереві, тому елементи завжди впорядковані, а пошук, вставлення й видалення виконуються за O(log n). Версії `multiset` і `multimap` допускають повторення ключів. Розв’яжіть класичні задачі: підрахунок частоти слів у тексті (`std::map<std::string, int>`), телефонний довідник, вилучення дублікатів.

Уважно розберіть доступ до `map`. `m[key]` **створює** елемент зі значенням за замовчуванням, якщо ключа немає, — зручно для лічильників, але

небезпечно під час простої перевірки. `m.at(key)` генерує `std::out_of_range`, якщо ключа немає. Для перевірки існування використовуйте `contains(key)` (C++20) або `find`. `try_emplace` додає елемент лише тоді, коли ключа ще немає. `lower_bound(key)` знаходить перший елемент, не менший за ключ, — так виконують пошук за діапазоном (наприклад, усі прізвища на літеру «К»).

Щоб зберігати в `set` або використовувати ключем у `map` власні об'єкти, потрібне правило порівняння: оператор `<` (або `<=>`) у класі чи **користувацький компаратор** — функціональний об'єкт, що передається третім аргументом шаблону: `std::set<Student, ByAvgDesc>`. Неупорядковані контейнери (`unordered_set`, `unordered_map`) побудовані на хеш-таблиці й працюють у середньому за O(1), але не зберігають порядку; для власного ключа їм потрібні хеш-функція (спеціалізація `std::hash<T>` або функтор) та операція `==`. Порівняйте швидкодію `map` і `unordered_map` на мільйоні вставлень і пошуків.

Адаптери надають обмежений інтерфейс поверх іншого контейнера: `std::stack` (LIFO — останнім прийшов, першим вийшов) для перевірки дужкових послідовностей і обчислення виразів у зворотному польському записі; `std::queue` (FIFO) для моделювання черги обслуговування; `std::priority_queue` — черга з пріоритетом, що завжди видає найбільший (або, з `std::greater`, найменший) елемент, — для планувальника задач.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити програму «Аналіз тексту»: прочитати текст із файлу, підрахувати частоту слів (`std::map`), вивести 10 найчастіших слів, усі унікальні слова в алфавітному порядку (`std::set`) і слова, що починаються на задану літеру (`lower_bound`). Створити клас `Student` з компаратором для `std::set` (за середнім балом, за спаданням) і хеш-функцією для `std::unordered_set`. Порівняти час вставлення та пошуку 1 000 000 цілих ключів у `std::map` і `std::unordered_map`. Реалізувати перевірку правильності дужкової послідовності `()[]{}` (`std::stack`) і планувальник завдань з

пріоритетами (`std::priority_queue`), що виконує задачі від найважливішої. Оформити порівняльну таблицю асоціативних контейнерів.

Запитання для самоперевірки:

1. Чим `std::map` відрізняється від `std::unordered_map` за будовою, складністю операцій і порядком елементів?
2. Чому для перевірки наявності ключа не варто використовувати `operator[]`?
3. Що потрібно визначити, щоб використовувати власний клас як ключ у `std::set` і в `std::unordered_set`?

ТЕМА 12. АЛГОРИТМИ STL. БІБЛІОТЕКА ДІАПАЗОНІВ

Література: *2, 8; інформаційні ресурси: 14, 16, 18, 23.*

Кількість годин для самостійної роботи: *6.*

Питання для самостійного вивчення:

1. Алгоритми без модифікації та зі зміною послідовності: find_if, count_if, transform, remove_if, unique, rotate, copy_if.
2. Алгоритми сортування й пошуку: sort, stable_sort, partial_sort, nth_element, binary_search, merge. Лямбда-вирази як компаратори.
3. Числові алгоритми: accumulate, reduce, iota, partial_sum, inner_product. Ітератори-адаптери.
4. Бібліотека діапазонів: views::filter, views::transform, views::take, конвеєри.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Бібліотека `<algorithm>` містить понад сто готових алгоритмів, які працюють з будь-якими контейнерами через ітератори. Замінюючи ручні цикли алгоритмами, ви отримуєте коротший, зрозуміліший і перевірений код: назва `count_if` одразу пояснює, що робить рядок, а цикл доводиться читати повністю. Візьміть кілька своїх попередніх програм і перепишіть цикли пошуку, підрахунку й перетворення через `find_if`, `count_if`,

`transform`, `copy_if`. Умову передавайте лямбда-виразом: `std::count_if(v.begin(), v.end(), [](int x) { return x > 0; });`.

Порівняйте алгоритми сортування: `sort` — найшвидший, але не зберігає порядок рівних елементів; `stable_sort` — зберігає (важливо під час сортування за кількома полями); `partial_sort` впорядковує лише перші k елементів (топ-10); `nth_element` ставить на місце лише один елемент (медіана). `binary_search`, `lower_bound`, `merge` працюють лише з **відсортованими** послідовностями. Компаратор має задавати строгий порядок (`<`, а не `<=`), інакше поведінка не визначена.

Опрацюйте `<numeric>`: `accumulate` (сума або згортка з довільною операцією; зверніть увагу на тип початкового значення — `0` дає цілочисельну суму навіть для `double`), `iota` (заповнення послідовними значеннями), `partial_sum` (накопичені суми), `inner_product` (скалярний добуток). Ітератори-адаптери `std::back_inserter` (додає елементи в кінець контейнера) та `std::ostream_iterator` (виводить елементи в потік) дозволяють поєднувати алгоритми без ручного виділення місця.

Бібліотека діапазонів C++20 спрощує запис: `std::ranges::sort(v);` замість пари ітераторів. **Представлення** (views) — це «ліниві» перетворення, які не копіюють дані: `v | std::views::filter(isEven) | std::views::transform(square) | std::views::take(5)` — конвеєр, що обчислює значення лише під час перебору. Порівняйте три реалізації однієї задачі (цикл, алгоритми STL, діапазони) за кількістю рядків, читабельністю і часом виконання.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити програму «Аналіз продажів»: вектор структур `Sale` (товар, категорія, кількість, ціна) заповнюється з файлу або випадково. За допомогою алгоритмів STL (без ручних циклів) обчислити: загальну виручку (`accumulate`), кількість продажів понад задану суму (`count_if`), список товарів категорії (`copy_if` + `back_inserter`), топ-5 продажів (`partial_sort`),

медіану ціни (`nth_element`), накопичену виручку по днях (`partial_sum`), унікальні категорії (`sort + unique`). Ту саму задачу «сума квадратів парних чисел серед перших 10, більших за 100» реалізувати циклом, алгоритмами та конвеєром `<ranges>` і порівняти час виконання на 10 млн елементів.

Запитання для самоперевірки:

1. Чим `sort` відрізняється від `stable_sort`? Коли важливо використовувати стабільне сортування?
2. Чому `accumulate(v.begin(), v.end(), 0)` для вектора `double` дає неправильний результат?
3. Що таке «ліниве» представлення в бібліотеці діапазонів і яку перевагу воно дає?

ТЕМА 13. РЯДКИ, ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ ТА ФАЙЛОВА СИСТЕМА

Література: *2, 4, 8; інформаційні ресурси: 14, 23, 26.*

Кількість годин для самостійної роботи: *4.*

Питання для самостійного вивчення:

1. std::string і std::string_view. Перетворення між рядками та числами (stoi, stod, from_chars, to_chars).
2. Форматування виведення маніпуляторами та std::format. Розбір рядків за допомогою std::stringstream.
3. Серіалізація об’єктів у текстових і двійкових файлах. Опрацювання CSV-файлів з обробкою помилок.
4. Бібліотека <filesystem>. Ознайомлення з <regex> та <chrono>.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

`std::string` **володіє** своїми символами, а `std::string_view` (C++17) — лише «вікно» на чужий рядок: покажчик і довжина без копіювання. Тому параметр функції, що лише читає текст, оголошуйте як `std::string_view`. Але пам’ятайте про небезпеку **висячого** `string_view`: не повертайте його з функції, якщо він посилається на локальний рядок, і не зберігайте

`string_view` на тимчасовий рядок. Для перетворень використовуйте `std::stoi`, `std::stod` (генерують виключення при помилці) або швидші `std::from_chars`/`std::to_chars`, що повертають код помилки замість виключення.

Функція `std::format` (C++20) поєднує зручність `printf` і безпеку потоків: `std::format("{:<10}|{:>8.2f}", name, price)` — ліве вирівнювання в полі 10 символів і число з двома знаками в полі 8. Вивчіть основні специфікатори: ширина, вирівнювання `<`, `>`, `^`, точність `.2f`, заповнення `{:*^20}`. `std::stringstream` дозволяє розбирати рядок так само, як консольне введення: `std::istringstream in(line); in >> name >> age;`, а `std::getline(in, field, ';')` розбиває рядок CSV на поля.

Серіалізація — збереження стану об’єкта у файл і відновлення з нього. Найпростіший спосіб — перевантажити для класу операції `<<` і `>>` та використовувати їх з `std::ofstream` і `std::ifstream`. Двійкові файли (`std::ios::binary`, методи `write`/`read`) компактніші, але придатні лише для простих типів без покажчиків і `std::string`. Під час читання CSV перевіряйте відкриття файлу, кількість полів у кожному рядку та коректність чисел, а пошкоджені рядки пропускайте з повідомленням, не перериваючи всю програму.

Бібліотека `<filesystem>` дає змогу працювати з файлами незалежно від ОС: `std::filesystem::exists(p)`, `file_size`, `create_directories`, обхід каталогу `directory_iterator`. Як ознайомлення розгляньте `<regex>` (перевірка формату e-mail чи дати за шаблоном) і `<chrono>` (вимірювання часу та робота з датами).

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити програму «Телефонна книга»: клас `Contact` (ім’я, телефон, e-mail, дата народження) з перевантаженими `<<` і `>>`; збереження та завантаження списку у CSV-файл з пропуском пошкоджених рядків і звітом про них; перевірка формату телефону та e-mail регулярними виразами;

функція пошуку з параметром `std::string_view`; виведення таблиці контактів за допомогою `std::format` з вирівнюванням стовпців. Додатково: утиліта, що за допомогою `<filesystem>` виводить усі файли заданого каталогу з розширенням `.csv`, їхній розмір і загальний обсяг, а також створює резервну копію каталогу в підпапці `backup`.

Запитання для самоперевірки:

1. Чим `std::string_view` відрізняється від `std::string`? У якій ситуації виникає висячий `string_view`?
2. Які переваги має `std::format` порівняно з маніпуляторами потоків?
3. Як реалізувати збереження та завантаження об'єктів власного класу у текстовий файл?

ТЕМА 14. БАГАТОПОТОКОВЕ ПРОГРАМУВАННЯ В C++

Література: *2, 8; інформаційні ресурси: 12, 14, 23.*

Кількість годин для самостійної роботи: *4.*

Питання для самостійного вивчення:

1. Створення потоків std::thread та std::jthread. Паралельне обчислення суми, пошук у кількох файлах.
2. Гонка даних. Синхронізація: std::mutex, std::lock_guard, std::scoped_lock. Взаємне блокування (deadlock).
3. Задача «виробник–споживач» і std::condition_variable. std::atomic.
4. std::async, std::future, std::promise. Вимірювання прискорення. ThreadSanitizer.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Потік — це окрема послідовність виконання всередині програми; кілька потоків виконуються одночасно на різних ядрах процесора. Об'єкт `std::thread t(func, args...)` запускає функцію в новому потоці, а `t.join()` очікує її завершення (якщо забути `join` або `detach`, програма аварійно завершиться). `std::jthread` (C++20) викликає `join` автоматично в деструкторі — це RAII для потоків. Розділіть обчислення суми великого

масиву на 4 частини, обчисліть кожну в окремому потоці та виміряйте прискорення для 1, 2, 4 і 8 потоків. У проєкті CMake підключіть бібліотеку потоків: `find_package(Threads REQUIRED)` і `target_link_libraries(app PRIVATE Threads::Threads)`.

Гонка даних виникає, коли кілька потоків одночасно змінюють ту саму змінну без синхронізації: інкремент `counter++` складається з трьох дій (прочитати, додати, записати), і потоки «затирають» результати один одного. Відтворіть це: 4 потоки по мільйону інкрементів дають менше ніж 4 000 000. Виправте двома способами: `std::mutex` з `std::lock_guard` (критична секція) та `std::atomic<int>` (атомарна операція) — і порівняйте час. Запустіть програму з ThreadSanitizer (`-fsanitize=thread`, у Compiler Explorer або на Linux) — він точно вкаже рядки з гонкою.

Взаємне блокування (deadlock) виникає, коли потік А тримає м'ютекс 1 і чекає на м'ютекс 2, а потік В — навпаки. Змоделюйте таку ситуацію на прикладі переказу між двома рахунками і усуньте її: захоплюйте кілька м'ютексів одночасно за допомогою `std::scoped_lock lock(m1, m2);` або завжди в однаковому порядку. Для координації потоків використовуйте `std::condition_variable`: споживач чекає (`wait`) на умову «черга не порожня», а виробник після додавання елемента сповіщає (`notify_one`).

Для задач, які просто повертають результат, зручніше високорівневе API: `auto f = std::async(std::launch::async, compute, data);` запускає функцію асинхронно і повертає `std::future`, а `f.get()` очікує та повертає результат (або повторно генерує виключення, що виникло в потоці). `std::promise` дозволяє передати значення з одного потоку в інший вручну. Пам'ятайте, що порядок виведення різних потоків непередбачуваний — це нормальна властивість багатопотокових програм.

ПРАКТИЧНЕ ЗАВДАННЯ

Розробити програму паралельної обробки даних: обчислити суму і максимум вектора з 50 млн чисел у 1, 2, 4 і 8 потоках (`std::jthread`) та вивести таблицю часу і прискорення; реалізувати ту саму задачу через

`std::async`/`std::future`. Відтворити гонку даних на спільному лічильнику та виправити її за допомогою `std::mutex` і `std::atomic`, порівнявши час. Змоделювати взаємне блокування під час переказу між рахунками та усунути його за допомогою `std::scoped_lock`. Реалізувати задачу «виробник–споживач»: два потоки-виробники генерують замовлення, один потік-споживач їх обробляє через чергу з `std::condition_variable`; програма завершується після обробки 20 замовлень.

Запитання для самоперевірки:

1. Що таке гонка даних і які засоби C++ дозволяють її усунути?
2. За яких умов виникає взаємне блокування і як його уникнути?
3. Чим `std::async` і `std::future` зручніші за ручне створення потоків?

ТЕМА 15. ПРИНЦИПИ ПРОЄКТУВАННЯ. ШАБЛОНИ ПРОЄКТУВАННЯ. МОДУЛЬНЕ ТЕСТУВАННЯ

Література: *1, 3, 8; інформаційні ресурси: 9, 14, 20, 30.*

Кількість годин для самостійної роботи: *4.*

Питання для самостійного вивчення:

1. Принципи SOLID, DRY і KISS. Аналіз і рефакторинг власного коду.
2. Шаблони проєктування: Singleton, Factory Method, Builder, Adapter, Decorator, Composite, Strategy, Observer, Command, State.
3. Модульне тестування з Google Test або Catch2: звичайні, межові та помилкові випадки, фікстури. Підключення тестів до CMake.
4. Документування коду коментарями Doxygen. Настанови C++ Core Guidelines.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Принципи **SOLID** допомагають створювати код, який легко змінювати. **S** (єдина відповідальність): клас має одну причину для змін — клас `Report` не повинен одночасно обчислювати дані, форматувати їх і записувати у файл. **O** (відкритість/закритість): нову поведінку додають новим класом, а не зміною наявного коду (`switch` за типом фігури — порушення). **L** (підстановка

Лісков): похідний клас має коректно працювати скрізь, де очікують базовий (класичний контрприклад — «Квадрат» як похідний від «Прямокутника»). **I** (розділення інтерфейсів): кілька вузьких інтерфейсів кращі за один «товстий». **D** (інверсія залежностей): класи залежать від абстракцій, а не від конкретних реалізацій. Проаналізуйте свій попередній проєкт і знайдіть щонайменше три порушення.

Шаблони проєктування — перевірені рішення типових задач. Поділяють їх на три групи: **породжувальні** (Singleton — єдиний екземпляр; Factory Method — створення об'єкта без знання конкретного класу; Builder — покрокова побудова складного об'єкта), **структурні** (Adapter — узгодження несумісних інтерфейсів; Decorator — додавання поведінки «обгорткою»; Composite — дерево об'єктів з однаковим інтерфейсом) і **поведінкові** (Strategy — змінний алгоритм; Observer — сповіщення підписників; Command — дія як об'єкт, зокрема для скасування; State — поведінка залежно від стану). Реалізуйте кожен шаблон коротким прикладом з власної предметної області та намалюйте його діаграму класів.

Модульний тест автоматично перевіряє окрему функцію або клас. У Catch2 тест записують так: `TEST_CASE("Deposit increases balance") { Account a(100); a.deposit(50); REQUIRE(a.balance() == 150); }`. Для кожного методу пишіть тести трьох видів: звичайні дані, межові значення (0, порожній контейнер, максимум) та помилкові (очікуване виключення: `REQUIRE_THROWS_AS(a.withdraw(-1), std::invalid_argument)`). Спільну підготовку виносьте у фікстуру або секції `SECTION`. Підключіть Catch2 до проєкту через `FetchContent` у `CMakeLists.txt`, створіть окрему ціль `tests` і запускайте їх командою `ctest` після кожної зміни коду.

Документуйте відкритий інтерфейс класів коментарями Doxygen (`///` `@brief`, `@param`, `@return`, `@throws`) — з них автоматично генерується HTML-документація. Ознайомтеся з розділами C++ Core Guidelines про класи (C), керування ресурсами (R) та обробку помилок (E) і порівняйте їх зі своїм кодом.

ПРАКТИЧНЕ ЗАВДАННЯ

Провести рефакторинг одного з власних проєктів відповідно до принципів SOLID: оформити в конспекті таблицю «порушення — принцип — виправлення» (не менше 4 рядків). Реалізувати систему «Інтернет-магазин» із застосуванням шаблонів Factory Method (створення товарів різних типів), Strategy (способи розрахунку знижки), Observer (сповіщення покупців про зміну ціни), Decorator (подарункова упаковка й доставка як «обгортки» замовлення) та Command (додавання/видалення товару з кошика з можливістю скасування). Написати не менше 15 модульних тестів Catch2 (звичайні, межові, помилкові випадки, фікстура), підключених до CMake, та документувати відкриті класи коментарями Doxygen.

Запитання для самоперевірки:

1. Розкрийте зміст кожного з принципів SOLID і наведіть приклад порушення одного з них.
2. Які задачі розв’язують шаблони Strategy, Observer і Factory Method?
3. Які групи тестових випадків потрібно передбачити для методу класу? Як перевірити, що метод генерує виключення?

ТЕМА 16. ПРАКТИКУМ З ПРОГРАМУВАННЯ: РОЗРОБЛЕННЯ ТА ЗАХИСТ ПРИКЛАДНОГО ПРОЄКТУ

Література: *1, 2, 3, 8; інформаційні ресурси: 9, 19, 20, 29.*

Кількість годин для самостійної роботи: *14.*

Питання для самостійного вивчення:

1. Вибір теми індивідуального проєкту, формулювання функціональних вимог і плану роботи.
2. Проєктування ієрархії класів, діаграми класів UML і структури модулів. Вибір контейнерів і алгоритмів STL.
3. Поетапна реалізація застосунку, модульне тестування, збирання CMake, ведення історії в Git.
4. Інструкція користувача, презентація та доповідь. Репетиція захисту.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Завершальна тема інтегрує все вивчене: інкапсуляцію, успадкування й поліморфізм, розумні вказівники, виключення, шаблони, контейнери й алгоритми STL, файлове введення-виведення, модульні тести. Почніть з вибору теми (перелік варіантів наведено в ПР № 35) і **технічного завдання**: мета програми, 8–12 функціональних вимог у формі «Користувач може …», формат файлів даних, обмеження. Узгодьте технічне завдання з викладачем до початку програмування.

Спроєктуйте програму: діаграма класів UML (базові абстракції, ієрархії, зв'язки композиції), розподіл на модулі (заголовкові файли й файли реалізації в папках `include`, `src`, `tests`), схема меню. Обґрунтуйте вибір кожного контейнера: де потрібен швидкий пошук за ключем (`map`/`unordered_map`), де важливий порядок (`vector` з сортуванням), де — черга чи стек. Перевірте дизайн на відповідність принципам SOLID ще до написання коду — виправити діаграму значно легше, ніж готову програму.

Реалізуйте проєкт ітераціями: каркас CMake з цілями `app` і `tests` → модель даних (класи) з тестами → збереження та завантаження → бізнес-логіка (пошук, фільтрація, сортування, звіти) → інтерфейс меню з перевіркою введення → опрацювання помилок. Після кожного етапу програма має збиратися без попереджень, проходити всі тести й фіксуватися в Git окремим комітом зі змістовним повідомленням. Наприкінці зробіть ревізію коду: видаліть дублювання, замініть ручні цикли алгоритмами, перевірте const-коректність і відсутність «сирих» `new`/`delete`.

Підготуйте супровідні матеріали: `README.md` (опис, вимоги до збирання, приклади використання), інструкцію користувача, коментарі Doxygen, звіт з таблицею тестів. Презентація (7–10 слайдів) має містити постановку задачі, діаграму класів, застосовані шаблони проєктування, демонстрацію роботи, результати тестування та висновки. Відрепетируйте доповідь на 5–7 хвилин і підготуйтеся пояснити будь-який фрагмент коду, обґрунтувати архітектурні рішення та відповісти, як програма поводиться в разі помилок.

ПРАКТИЧНЕ ЗАВДАННЯ

Виконати індивідуальний проєкт за погодженою темою (наприклад, «Система управління бібліотекою», «Облік замовлень інтернет-магазину», «Планувальник завдань команди», «Каталог фільмів з рейтингами», «Облік тренувань»). Обов’язкові вимоги: ієрархія класів з віртуальними методами та розумними вказівниками; щонайменше один власний шаблон класу або функції; використання контейнерів і алгоритмів STL (зокрема лямбда-виразів); власна ієрархія виключень; збереження та завантаження даних у файл; збирання CMake з цілями застосунку й тестів; не менше 10 модульних тестів Catch2; історія розроблення в Git (не менше 10 змістовних комітів); README.md, інструкція користувача та презентація для захисту.

Запитання для самоперевірки:

1. Які етапи проходить розроблення програмного проєкту від ідеї до захисту?
2. Як ви обґрунтуєте вибір контейнерів STL і шаблонів проєктування у своєму проєкті?
3. Які матеріали має містити документація проєкту?

КРИТЕРІЇ ОЦІНЮВАННЯ ВИКОНАННЯ САМОСТІЙНОЇ РОБОТИ

Оцінювання результатів самостійної роботи студента здійснюється під час поточного контролю у формі перевірки виконаних практичних завдань (програмного коду та результатів його роботи), усного опитування за запитаннями для самоперевірки та перевірки конспектів опрацьованих тем.

Під час оцінювання враховується: повнота опрацювання теоретичних питань; працездатність і правильність розробленого програмного коду (проєкт збирається без помилок і попереджень, дає правильні результати, проходить модульні тести, не має витоків пам’яті); доцільність застосування об’єктно-орієнтованих конструкцій, засобів стандартної бібліотеки та сучасних можливостей C++; якість оформлення та документування коду; наявність історії змін у Git; самостійність виконання завдань; уміння пояснити архітектурні рішення та логіку власного коду.

Програмний код, який студент не може пояснити, або код, ідентичний роботам інших студентів, не зараховується. Практичне завдання з теми 16 (індивідуальний проєкт) оцінюється під час його захисту.

Наявність виконаної самостійної роботи за всіма темами (оформлених практичних завдань та відповідей на запитання для самоперевірки) є обов’язковим компонентом для отримання максимального поточного семестрового балу та допуску до підсумкового диференційованого заліку.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ ТА ІНФОРМАЦІЙНИХ РЕСУРСІВ

Основна література

1. Зеленський О. С., Лисенко В. С. Об’єктно-орієнтоване програмування на C++ : навч. посіб. Кривий Ріг : ДУЕТ, 2023. 215 с.
2. Міловідов Ю. О. Об’єктно-орієнтоване програмування : навч. посіб. 3-тє вид., допов. і переробл. Київ : НУБіП України, 2025. 334 с.
3. C++. Алгоритмізація та програмування : підручник / О. Г. Трофименко та ін. 2-ге вид., перероб. і допов. Одеса : Фенікс, 2019. 477 с.

Додаткова література

4. Васильєв О. Програмування на C++ в прикладах і задачах : навч. посіб. Київ : Ліра-К, 2017. 382 с.
5. Про затвердження стандарту фахової передвищої освіти зі спеціальності F2 «Інженерія програмного забезпечення» галузі знань 12 «Інформаційні технології» для освітньо-професійного ступеня «фаховий молодший бакалавр» : наказ МОН України від 21 верес. 2021 р. № 1006. URL: https://osvita.ua/legislation/faxova-peredvyshha-osvita/85816/ (дата звернення: 29.10.2025).
6. Про фахову передвищу освіту : Закон України від 6 черв. 2019 р. № 2745-VIII. URL: https://zakon.rada.gov.ua/laws/show/2745-19 (дата звернення: 12.11.2025).
7. Stroustrup B. Programming: Principles and Practice Using C++. 3rd ed. Boston : Addison-Wesley, 2024. 656 p.

Інформаційні ресурси

8. C++ Core Guidelines : вебсайт. URL: https://isocpp.github.io/CppCoreGuidelines/ (дата звернення: 31.10.2025).
9. C++ Insights : вебсайт. URL: https://cppinsights.io (дата звернення: 15.11.2025).
10. Codewars : вебсайт. URL: https://www.codewars.com (дата звернення: 06.11.2025).
11. Compiler Explorer (Godbolt) : вебсайт. URL: https://godbolt.org (дата звернення: 26.09.2025).
12. cplusplus.com : вебсайт. URL: https://cplusplus.com (дата звернення: 25.09.2025).

13. cppreference.com : вебсайт. URL: https://en.cppreference.com (дата звернення: 11.11.2025).
14. e-olymp : вебсайт. URL: https://www.e-olymp.com (дата звернення: 01.11.2025).
15. Exercism. C++ Track : вебсайт. URL: https://exercism.org/tracks/cpp (дата звернення: 09.11.2025).
16. freeCodeCamp : вебсайт. URL: https://www.freecodecamp.org (дата звернення: 14.09.2025).
17. GeeksforGeeks. C++ : вебсайт. URL: https://www.geeksforgeeks.org/c-plus-plus/ (дата звернення: 14.11.2025).
18. Git : вебсайт. URL: https://git-scm.com/doc (дата звернення: 21.09.2025).
19. GitHub : вебсайт. URL: https://github.com (дата звернення: 11.09.2025).
20. GNU Compiler Collection (GCC) : вебсайт. URL: https://gcc.gnu.org (дата звернення: 07.09.2025).
21. HackerRank. C++ : вебсайт. URL: https://www.hackerrank.com/domains/cpp (дата звернення: 06.10.2025).
22. LearnCpp.com : вебсайт. URL: https://www.learncpp.com (дата звернення: 23.10.2025).
23. LeetCode : вебсайт. URL: https://leetcode.com (дата звернення: 28.10.2025).
24. OnlineGDB : вебсайт. URL: https://www.onlinegdb.com (дата звернення: 16.10.2025).
25. Programiz. Learn C++ Programming : вебсайт. URL: https://www.programiz.com/cpp-programming (дата звернення: 12.10.2025).
26. Prometheus : вебсайт. URL: https://prometheus.org.ua (дата звернення: 04.09.2025).
27. Replit : вебсайт. URL: https://replit.com (дата звернення: 10.10.2025).
28. Stack Overflow : вебсайт. URL: https://stackoverflow.com (дата звернення: 27.09.2025).
29. Standard C++ Foundation : вебсайт. URL: https://isocpp.org (дата звернення: 30.09.2025).
30. Visual Studio Code : вебсайт. URL: https://code.visualstudio.com/docs (дата звернення: 13.10.2025).
31. W3Schools. C++ Tutorial : вебсайт. URL: https://www.w3schools.com/cpp/ (дата звернення: 19.10.2025).