Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
”КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ”

ФАКУЛЬТЕТ ІНФОРМАТИКИ
Кафедра мультимедійних систем

КВАЛІФІКАЦІЙНА РОБОТА
освітній ступінь – бакалавр

на тему: **”Розвиток стратегій проектування Александреску. Бібліотека Loki23”**

Виконав: студент 4-го року навчання,
Освітньої програми «Інженерія програмного забезпечення», 121
Трохимчук Артем Андрійович

Керівник: Бублик В.В.,
кандидат фіз.-мат. наук, доцент

Київ – 2026

АНОТАЦІЯ

У роботі розглянуто поняття стратегій проектування Александреску та засоби, якими цей підхід може бути реалізований у сучасному C++. Досліджено бібліотеку `Loki` як приклад використання стратегій проектування, шаблонного метапрограмування та узагальненого програмування в C++98, а також її вплив на розвиток Modern C++ і стандартної бібліотеки. Було розроблено бібліотеку мовою C++23, що містить модернізовані реалізації основних компонентів `Loki`: списків типів, генерації ієрархій, фабрик об'єктів, розумного вказівника, `Singleton` і мультиметодів. Як приклад використання створеної бібліотеки був реалізований застосунок `netprobe`, який демонструє переваги та обмеження запропонованих абстракцій під час аналізу мережевого трафіку.

Ключові слова: стратегії проектування, Modern C++, C++23, `Loki`, `Loki23`, шаблонне метапрограмування.

ЗМІСТ

ВСТУП **5**

1 РОЗДІЛ 1. Теоретичні основи стратегій проектування Александреску **7**

1 Концепція проектування на основі стратегій 7

2 Узагальнене програмування та шаблонне метапрограмування як база для бібліотеки Loki 10

3 Огляд класичних компонентів Loki: SmartPtr, TypeList та Factory 12

3.1 Управління ресурсами в гнучкий спосіб 13

3.2 Революція в маніпуляції метаданими 14

3.3 Переосмислення фабрик об’єктів 15

4 Проблеми та обмеження оригінальної реалізації в контексті стандартів C++98/03 15

2 Технологічний стек C++20/23 як інструмент модернізації **18**

1 Роль концептів у заміні SFINAE та покращенні діагностики помилок 18

2 Використання constexpr та consteval для перенесення обчислень у стадію компіляції 20

3 Варіативні шаблони та Fold Expressions у сучасних стратегіях . 22

4 Переваги модульної системи для архітектури бібліотек шаблонів 25

3 Вплив Loki на Modern C++ **27**

1 Бібліотека метапрограмування та перевірки часу компіляції в Modern C++ 27

2 Функтори Modern C++ як нащадки Loki 28

3 Еволюція розумних вказівників у C++ 30

4 Архітектура та реалізація бібліотеки Loki23 **32**

1 Переосмислення ієрархії стратегій за допомогою обмежень та концептів 32

2 Реалізація сучасного розумного вказівника з підтримкою атомарних операцій та власних стратегій володіння 35

3 Модернізація патерну Singleton: потокобезпека та життєвий цикл об’єктів у C++23 37
4 Реалізація ієрархій класів 40
5 Мультиметоди в Modern C++ 41

5 Практичне застосування та порівняльний аналіз 45
1 Розробка тестового застосунку з використанням компонентів Loki23 45
2 Порівняння продуктивності оригінальної Loki та Loki23 50
2.1 Результати 52
2.2 Аналіз результатів 53
3 Аналіз чистоти коду та зручності використання 54

ВИСНОВКИ 58

Список використаних джерел 60

ВСТУП

Процес розробки програмного забезпечення складається з багатьох етапів: узгодження вимог, проектування архітектури, кодування. І хоча процес написання коду не є найбільш відповідальним чи складним, ним не можна нехтувати. Якість написаного коду безпосередньо впливає на кінцевого користувача: програма може мати бездоганну архітектуру, чітко сформульовані й виконані вимоги, проте запускатиметься на комп'ютері користувача півтори хвилини, споживатиме всю наявну оперативну пам'ять та падатиме кожні півгодини. Якість програми безпосередньо залежить від якості коду. Це логічно і зрозуміло: складно зробити якісну роботу неякісними матеріалами. Як виявляється, якість коду також залежить від мови програмування [9]. Програми, написані на C, "течуть" частіше за програми на JavaScript не тому, що їх гірше проектують чи розробники гірше пишуть код: на це впливає сама мова програмування. Таким чином, чим кращі абстракції надає мова програмування та її стандартна бібліотека, тим вища якість кінцевої програми.

У порівнянні з можливостями мови стандартну бібліотеку C++ можна назвати аскетичною: лише в C++20 для множин було реалізовано метод `contains()` для перевірки, чи є елемент у наборі. У 2001 році Андрей Александреску спробував виправити цю проблему: він написав `Loki` — бібліотеку, яка додає в C++ абстракції, що, на його думку, були корисні кожному розробнику, але яких досі не було в стандартній бібліотеці. Ця бібліотека стала в прямому сенсі доленосною для C++: багато абстракцій, запропонованих Александреску, згодом було стандартизовано в C++.

Актуальність теми зумовлена передусім змінами в C++: порівнюючи C++98 та C++23, можна навести приклад лампової ЕОМ, яка займала поверх будинку, і сучасного комп'ютера. Виведення типів, розумні вказівники, розширення шаблонного метапрограмування, яким вдалося замінити важке використання препроцесора, а також нові можливості стандартної бібліотеки шаблонів — усе це дозволяє говорити про Modern C++ як про окремий етап розвитку мови. Ба більше, зміни відбулись і у свідомості розробників: якщо C++98 можна було вважати "C з класами", то в сучасному C++ стався перехід від процедурно-імперативного стилю до мультипарадигменності й використання функціонального програмування. Варто також зауважити, що значною мірою це заслуга Андрея Александреску і вищезгаданої бібліотеки.

Метою роботи є дослідження Modern C++ як принципово нової мови на прикладі змін до бібліотеки `Loki` і написання `Loki23` — сучасної бібліотеки абстракцій для C++.

Завдання роботи:

1. Дослідити реалізацію бібліотеки Loki Андрея Александреску, її можливості й вплив на розвиток C++
2. Виділити засоби Modern C++ за їх походженням від бібліотеки Loki
3. Реалізувати бібліотеку Loki23 як модернізацію бібліотеки Loki із використанням можливостей Modern C++

Об'єктом дослідження є Modern C++ як логічний розвиток C++, а також шаблонне метапрограмування як невід'ємна частина C++.

Предметом дослідження є бібліотека `Loki23` як приклад використання сучасного C++ для модернізації коду legacy C++, а також використання шаблонного метапрограмування для заміни препроцесора і вплив бібліотеки `Loki` на розвиток C++

Практичне значення роботи полягає у документуванні розвитку C++ від C++98 до сучасних редакцій C++ на прикладі бібліотеки `Loki`. Також практичне значення має написання бібліотеки `Loki23` із реалізацією абстракцій, яких досі бракує в STL.

Наукова новизна роботи полягає в подальшому розвитку підходу Александреску до проектування на основі стратегій у контексті Modern C++: у роботі показано, що ключові абстракції `Loki` можуть бути перенесені з обмежень C++98/03 у модель C++20/23 із варіативними шаблонами, концептами, модулями, семантикою переміщення та стандартними засобами метапрограмування. Запропонована реалізація `Loki23` формалізує раніше неявні вимоги до стратегій і типів, замінює макросні та рекурсивні механізми декларативнішими конструкціями мови.

РОЗДІЛ 1. РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ СТРАТЕГІЙ ПРОЕКТУВАННЯ АЛЕКСАНДРЕСКУ

1. Концепція проектування на основі стратегій

Однією з центральних ідей, запропонованих Андреєм Александреску в книзі Modern C++ design, є використання стратегій для керування поведінкою класів. Проблема, яку вирішують стратегії, не нова: яким чином можна вплинути на поведінку екземпляра класу? Наприклад, розглянемо задачу побудови класу розумного вказівника `SmartPtr`, який може використовувати лічильник посилань (як у `std::shared_ptr`) для видалення об'єктів або сторонній збирач сміття.

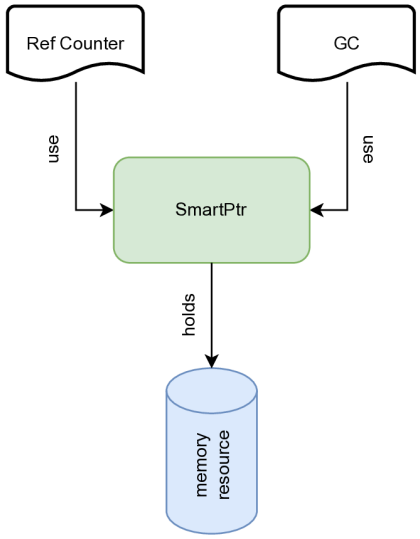

Рис. 1: Логічна будова SmartPtr

Здавалося б, об'єктно-орієнтоване програмування вже надає спосіб розв'язувати цю проблему:

1. Ми робимо `SmartPtr` абстрактним класом, у якому функціональність підрахунку посилань винесена в абстрактні функції.
2. У класах `CountedSmartPtr` та `GCSmartPtr` ми надаємо реалізацію цих методів.

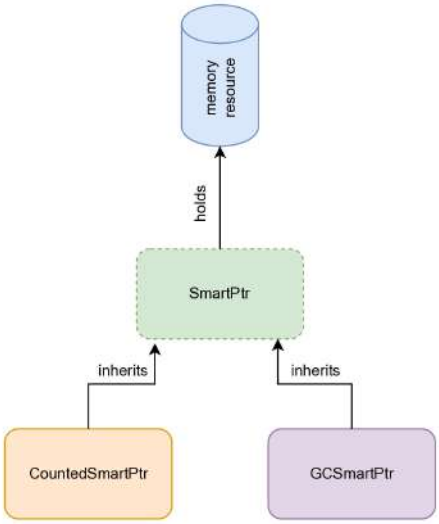

Рис. 2: Логічна будова SmartPtr в об’єктно-орієнтованому стилі

І, здавалося б, цей підхід працює, ба більше, він використовується в більшості сучасних об’єктно-орієнтованих мов програмування для розв’язання схожих задач.

Але використання ООП має одну дуже неприємну особливість: ми маємо визначити всі класи завчасно. Тобто якщо таких класів може бути 3, ми маємо визначити всі 3 класи. Цю проблему складно побачити у доволі простому прикладі із розумним вказівником, але якби ми мали, наприклад, 3 програмовані властивості та по 2 варіанти поведінки на кожну, ми б мали надати реалізацію 2^3 класів. Окремим завданням для розробника було б реалізувати ту саму стратегію видалення в усіх класах без повторення коду.

Було б несправедливо говорити про об’єктно-орієнтоване програмування лише в контексті наслідування. Ту саму задачу можна було б реалізувати із використанням композиції: замість того щоб робити клас `SmartPtr` абстрактним і наслідуватись від нього, можна винести програмовані фрагменти коду в окремі класи і приймати реалізацію цих класів у конструкторі. Таким чином нам не треба буде надавати N реалізацій класів для всіх можливих комбінацій поведінок.

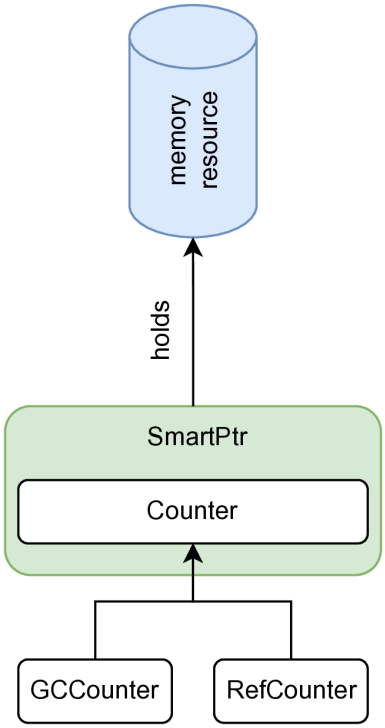

Рис. 3: Логічна будова SmartPtr із використанням композиції

Проте такий підхід створює принципову проблему: два абсолютно різні об'єкти `SmartPtr` з різними поведінками матимуть один і той самий тип (статичний та динамічний), тому для користувачів вони виглядатимуть як один і той самий тип. Це призводить до того, що в усіх бінарних операторах необхідно робити перевірку на ідентичність стратегій. Таким чином створюється система типів поверх тієї, що вже існує в C++, ця система типів повністю ручна, тобто програмована користувачем. Зростає ймовірність помилки, і це просто незручно. Але попри це, композиція виглядає саме як програмування поведінки: у вас є якісь "блоки", що описують, що клас має робити в тій чи іншій ситуації. Сам же клас `SmartPtr` слугує каркасом, який робить загальні речі (наприклад виділяє пам'ять для ресурсу чи зберігає його адресу) і викликає програмовані частинки в необхідний момент.

Стратегії класів, запропоновані Андреєм Александреску, дозволяють реалізувати подібну до композиції логіку винесення програмованих фрагментів в окремі класи, але з урахуванням цього в типі, тобто поведінка класу прямо вноситься в статичний тип. Таким чином в усіх операторах за набором стратегій буде слідкувати компілятор, а не користувач. Тобто стратегія — це програмована частина класу, яка є частиною статичного типу цього класу. Використовуючи мову шаблонів C++, стратегії дозволяють

перевикористати код класу і записати інформацію про те, яким чином клас був ”сконфігурований”, у метадані класу (по суті в статичний тип класу).

Порівнюючи з об’єктно-орієнтованим дизайном, проектування на основі стратегій може бути дещо незвичним. Якщо порівнювати із наслідуванням, класи, які реалізують програмовану логіку, володіють усім контекстом. Тобто клас, який відповідає за підрахунок посилань на ресурс, може взагалі не знати, про який ресурс ідеться: вказівник чи файловий дескриптор. Його завдання — рахувати посилання і казати, коли звільняти пам’ять. Такий підхід дуже схожий на композицію. Але стратегії набагато гнучкіші: по-перше, ніхто не забороняє використовувати лише клас стратегій, а не створювати його екземпляри. Тобто стратегія може мати лише статичні змінні та методи. По-друге, стратегія — це шаблонний аргумент, це означає, що клас не обов’язково має виконувати якийсь інтерфейс де-юре чи наслідувати абстрактний клас. Ну і звичайно, стратегії не створюють накладних витрат під час виконання: при наслідуванні обов’язково створюються віртуальні методи (як мінімум один метод завжди має бути віртуальним — деструктор). Віртуальні методи сповільнюють виконання програми, і, що ще гірше, переносять частину перевірок типів на час виконання. Натомість стратегії Александреску повністю перевіряються компілятором один раз під час збирання програми.

2. Узагальнене програмування та шаблонне метапрограмування як база для бібліотеки Loki

Узагальнене програмування — це парадигма програмування, що полягає в описі алгоритмів чи структур даних, *абстрагованих* від конкретних типів. Проблема, яку воно вирішує, очевидна: мова програмування оперує конкретними типами даних, але алгоритми частіше за все оперують цілими класами типів даних.

Візьмемо за приклад теорему Піфагора для пошуку довжини гіпотенузи прямокутного трикутника за двома катетами:

$$c = \sqrt{a^2 + b^2}$$

У математиці a, b, c — числа, проте в програмуванні ми оперуємо даними якогось типу, наприклад тип `int` в C++ має щонайменше 16 біт і покриває

значення щонайменше в діапазоні $[-32768; 32767]$, а `unsigned` при тій самій кількості біт покриває значення в діапазоні $[0; 65535]$. Від зміни типу алгоритм Піфагора *ніяк* не змінився, незалежно від того, використовується `int`, `unsigned` чи навіть `float`. Саме такі алгоритми без прив'язки до конкретного типу й дозволяє реалізувати узагальнене програмування. В C++ цей підхід реалізується через шаблони, за допомогою яких сам тип операндів стає аргументом. А об'єктний код для кожного набору шаблонних аргументів генерує сам компілятор, під час збирання програми.

Мова шаблонів — напевно, одна з найвиразніших частин C++, чого вартий лише той факт, що це повна за Тьюрингом функціональна мова програмування в межах C++. Хоча мова шаблонів початково мала б бути лише реалізацією узагальненого програмування в C++, вона виявилася одним із головних механізмів для реалізації рефлексії та обчислень часу компіляції в C++, тобто універсальним механізмом метапрограмування.

Спочатку слід розібратися, що взагалі таке метапрограмування в C++? Метапрограмування в C++ — це написання коду, який виконується під час компіляції програми. Оскільки об'єктний код за мовою шаблонів генерується під час компіляції, то й програми, написані цією мовою, "виконуються" лише під час компіляції. Таким чином компілятор стає середовищем виконання коду (по суті інтерпретатором мови шаблонів), а шаблони — **мовою програмування** цього середовища. Результат роботи такого коду — нові типи та нові функції, які вбудовуються у виконуваний файл вже в готовому вигляді. Але оскільки метапрограмування можливе не лише з типами, а й з константами, за допомогою мови шаблонів можна писати програми, що будуть виконані під час компіляції. Тобто використовуючи мову узагальненого програмування в C++ можна обчислити вищезгадану довжину гіпотенузи за теоремою Піфагора.

Метапрограмування в C++ ґрунтується на двох усталених практиках: SFINAE та спеціалізації шаблонів.

SFINAE — це абревіатура, що розшифровується як "Substitution Failure Is Not An Error" (невдала підстановка [шаблонного параметра] не є помилкою). На перший погляд, це звучить неочевидно, адже невдала підстановка мала б викликати помилку, вона, власне, невдала. Але замість генерації помилки компілятор *мовчки* прибирає цю *невдалу* підстановку з множини

усіх можливих кандидатів. Таким чином некоректні підстановки не заважають компілятору перевірити всі можливі варіанти і знайти коректну. Часто SFINAE використовується як спосіб вмикати чи вимикати шаблонний код залежно від аргументів. Наприклад, шаблонні аргументи функції обчислення гіпотенузи можна було б обмежити числовими типами: немає сенсу обчислювати гіпотенузу з двох рядків. Хоча компілятор цього й так не дозволить (внаслідок відсутності функції квадратного кореня для рядків), такі обмеження є частиною публічного інтерфейсу і дозволяють зробити повідомлення про помилки зрозумілішими.

Спеціалізація шаблону — це спосіб змінити шаблон для якихось конкретних аргументів. По суті, кожна спеціалізація — це виняткова реалізація узагальненого алгоритму. Найочевиднішим прикладом спеціалізації шаблонів є `std::vector<bool>`: замість того, щоб зберігати масив із елементами типу `bool` по 1 байту, з якого використовувався б лише 1 біт, більшість реалізацій STL використовує бітові поля задля економії пам'яті. Спеціалізація буває повна, коли надається реалізація для одного конкретного повного набору шаблонних параметрів, та часткова, коли описується *клас* спеціалізацій, у якому лише частина параметрів зафіксована.

Андрей Александреску використав можливості мови шаблонів на повну: неможливо уявити бібліотеку `Loki` без узагальненого програмування. Найочевидніше її використання — вищезгадані стратегії. В них шаблони використовуються за прямим призначенням узагальненого програмування: як аргументи для інших типів. Стратегії слугують метаданими класу, частиною його статичного типу і дозволяють гнучко впливати на поведінку класу.

Окрім використання мови шаблонів, `Loki` її розширює, надаючи реалізацію бібліотеки характеристик типів `TypeTraits`. Використовуючи її, користувач може дізнатись, чи є тип вказівником, посиланням, константою тощо. Це виявилось настільки зручно, що бібліотека характеристик типів була пізніше додана в STL C++.

3. Огляд класичних компонентів Loki: SmartPtr, TypeList та Factory

Бібліотека `Loki` — це, напевно, найкращий приклад того, як мова шаблонів може бути використана для реалізації узагальнених, виразних, безпечних

абстракцій. Найкраще це можна продемонструвати на прикладі розумних вказівників, списку типів і реалізації фабрик об'єктів.

3.1. Управління ресурсами в гнучкий спосіб

Управління ресурсами, можливо, один із найскладніших аспектів у сучасному програмуванні, тому є так багато видів управління ресурсами: від ручних алокацій та звільнення пам'яті в C, до окремих програм, які відповідають за алокацію і вчасне звільнення пам'яті в більшості інтерпретованих мов. Проблему управління динамічною пам'яттю в сучасному C++ закривають стандартизовані розумні вказівники `std::unique_ptr` та `std::shared_ptr`, які, на жаль, зовсім не гнучкі. Наприклад, `std::shared_ptr` — це єдина реалізація розумного вказівника в стандартній бібліотеці із сумісним (множинним) доступом до ресурсу. Він влаштований доволі просто: поруч із вказівником на об'єкт лежить лічильник посилань. Як тільки лічильник стає рівним 0, цей об'єкт у пам'яті видаляється. Стандарт вимагає від нього бути потокобезпечним, це означає, що збільшення чи зменшення використовує доволі дорогі атомарні операції над даними:

```
; ...
lock add DWORD PTR [rax],edx
; ...
```

Лістинг 1: Атомарна операція збільшення лічильника посилань у `std::shared_ptr`

Але що робити, якщо в програмі лише один потік? Чи можна якось нівелювати накладні витрати, зумовлені атомарними операціями? Виявляється, що ні: стандартна реалізація розумних вказівників недостатньо гнучка, щоб задовольнити всі потреби, ба більше, вона не пропонує фундаменту для побудови чогось більш гнучкого.

Але Александреску застосував для розв'язання задачі керування динамічною пам'яттю дизайн на базі стратегій: він розділив реалізацію розумного вказівника на загальну частину і декілька програмованих характеристик, винесених у стратегії:

1. `StoragePolicy` визначає, що саме зберігається (сирий вказівник, handle або масив) і як здійснюється доступ до об'єкта. Це дозволяє

використовувати SmartPtr навіть для роботи з ресурсами операційної системи, які не є вказівниками в чистому вигляді, наприклад із файловими дескрипторами.

2. OwnershipPolicy реалізує механізми управління володінням ресурсом. Варіанти включають підрахунок посилань (RefCounted), зв'язні списки (RefLinked), глибоке копіювання (DeepCopy) або деструктивне копіювання в стилі auto_ptr.

3. CheckingPolicy налаштовує рівень безпеки під час роботи із ресурсом, по суті захист від NULL. Можна обрати перевірку на NULL при кожному розіменуванні (RejectNull), лише у налагоджувальних збірках (AssertCheck) або взагалі відмовитися від перевірок для максимальної швидкості (NoCheck).

4. ConversionPolicy визначає, чи може розумний вказівник неявно приводитися до сирого вказівника, що часто є джерелом помилок, але необхідно для сумісності зі старим кодом на C++, написаним без використання Loki, чи з кодом на C.

3.2. Революція в маніпуляції метаданими

TypeList — це, мабуть, найбільш концептуально новий інструмент, представлений Александреску. Він дозволяє об'єднувати довільну кількість типів у структуру, з якою можна працювати за допомогою алгоритмів, подібних до тих, що застосовуються до звичайних списків у STL. Списки типів реалізовані як зв'язний список, що складається з пари: голови (тип) та хвоста (усі інші типи у списку). Для зручності використання Александреску запропонував систему макросів (від TYPELIST_1 до TYPELIST_50), які приховують рекурсивне визначення за плоским списком аргументів:

```
template <class T, class U>
struct Typelist
{
  typedef T Head;
  typedef U Tail;
}

```

```
typedef TYPELIST_3(int, double, char) TypeListExample1;
// Same as
Typelist<int,
  Typelist<double,
    Typelist<char, NullType >>> TypeListExample2;
```

Лістинг 2: Рекурсивне визначення списку типів у класичній `Loki`

3.3. Переосмислення фабрик об'єктів

`Loki` запропонувала доволі незвичне рішення для динамічного створення об'єктів. Традиційна проблема об'єктно-орієнтованого проектування полягає в тому, що для створення об'єкта певного типу у фабриці потрібно зробити дві речі:

1. Додати новий метод в інтерфейс фабрики.
2. Додати реалізацію інтерфейсу *в усі попередні* фабрики (щоб вони виконували новий інтерфейс).

Замість цього абстрактна фабрика за Александреску пропонує використати список типів для того, щоб зберігати, які об'єкти створює фабрика. Клас `GenScatterHierarchy`, розглянутий пізніше, створює структуру, де клас наслідує окремий шаблон для кожного типу зі списку, створюючи розгалужену ієрархію, яка слугує *інтерфейсом* фабрики об'єктів. `GenLinearHierarchy` створює лінійний ланцюжок наслідування, що використовується як реалізація фабрики за тим самим інтерфейсом.

Таким чином, замість того, щоб вручну описувати методи `CreateFoo`, `CreateBar` тощо, розробник передає список базових класів у шаблон `AbstractFactory`, і `Loki` самостійно генерує ієрархію віртуальних функцій для створення цих типів. Реалізація `ConcreteFactory` автоматизується, що гарантує цілісність сімейств створюваних об'єктів, зменшуючи шанс помилки через людський фактор.

4. Проблеми та обмеження оригінальної реалізації в контексті стандартів C++98/03

Свого часу бібліотека Loki викликала фурор у сфері програмування на C++, її реалізація в межах стандартів C++98/03 стикалася з серйозними

технічними перешкодами, які сьогодні здаються архаїчними, але в той час значно ускладнювали як реалізацію бібліотеки, так і її використання.

Найбільш помітним обмеженням C++98 була неможливість визначити шаблон із довільною кількістю параметрів. Це змусило Александреску використовувати обхідні шляхи:

1. Замість звичного варіативного шаблону типу `Template<T1, T2, T3>` розробникам доводилося використовувати списки типів і писати рекурсивні структури накшталт `Template<TypeList<T1, TypeList<T2, ... >>>`.

2. Для спрощення синтаксису Александреску впровадив макроси `TYPELIST_1, TYPELIST_2, ..., TYPELIST_50`. Це створювало штучне обмеження в 50 елементів та робило повідомлення про помилки компілятора майже нечитабельними через величезну глибину вкладеності.

Використання макросів є проблематичним з кількох причин. По-перше, макроси не є типобезпечними й не поважають простори імен, адже вони підставляються препроцесором до аналізу типів. По-друге, максимальна арність жорстко обмежена кількістю визначених макросів: для списку з 51 типом необхідно додати новий макрос. По-третє, повідомлення про помилки у разі некоректного використання макросів не є дружніми до користувача, оскільки компілятор бачить лише розгорнутий код і не бачить, що він походить з макроса. На щастя, стандарт C++11 повністю розв’язав цю проблему завдяки варіативним шаблонам.

Другою важливою проблемою була відсутність семантики переміщення в C++98. Взагалі не існувало концепції rvalue-посилань та семантики переміщення як такої. Єдиним механізмом передачі володіння ресурсом було деструктивне копіювання — підхід, реалізований у `std::auto_ptr`, де конструктор копіювання модифікує джерело, обнуляючи його вказівник. Стратегія `DestructiveCopy` у SmartPtr `Loki` страждала від тієї самої проблеми: прийняття неконстантного посилання у конструкторі копіювання унеможливлювало коректне повернення таких вказівників із функцій та зберігання у контейнерах. Семантика переміщень, визначена в стандарті C++11, вирішує цю проблему: тепер якщо треба вказати, що *власність*

над ресурсом передається, слід використовувати посилання на `rvalue`, а копіювання, як би очевидно це не звучало, означає копіювання, без модифікації того, що копіюється.

Але найбільш неочевидним недоліком `Loki` і, власне, того періоду програмування на C++ були компілятори, яких було багато і кожен з яких підтримував ”свою версію” C++. `Loki` фактично була не однією бібліотекою, а цілими 4 бібліотеками: одна для компілятора Borland, дві для MSVC (версій 1200 і 1300) і одна стандартна, для компіляторів, які працюють за стандартом. Порівнюючи реалізацію для MSVC зі стандартною, можна побачити, що цей компілятор вимагав повернення значень із функції завжди, навіть якщо сталася виняткова ситуація:

```
- throw Exception();
+ throw Exception();
+ return (AbstractProduct*) 0;
```

Лістинг 3: Відмінність реалізації `Loki` для MSVC із додатковим поверненням значення

Складно навіть уявити, як це підтримувати. На щастя, зараз ситуація набагато простіша: замість того, щоб підганяти реалізацію під компілятори, останні розробляються за стандартом. І не треба підтримувати набір реалізацій тієї самої бібліотеки.

РОЗДІЛ 2. ТЕХНОЛОГІЧНИЙ СТЕК C++20/23 ЯК ІНСТРУМЕНТ МОДЕРНІЗАЦІЇ

Починаючи зі стандарту C++11, мова програмування C++ отримала велику кількість оновлень: від варіативних шаблонів у мові шаблонів, семантики переміщення, яка дозволила підвищити швидкодію програми без її зміни [6], до розумних вказівників та мультипоточності в стандартній бібліотеці. Таким чином, C++ після C++11 варто розглядати *як принципово іншу мову*, Modern C++, яка вимагає переходу від імперативного підходу до написання коду в стилі "C з класами" до мультипарадигменного *програмування* із використанням передових технік метапрограмування.

1. Роль концептів у заміні SFINAE та покращенні діагностики помилок

Одним із найскладніших аспектів оригінальної бібліотеки Loki була необхідність використання механізму SFINAE для реалізації перевірок властивостей типів на етапі компіляції. Александреску використовував `sizeof()` разом із перевантаженими функціями для перевірки конвертованості типів та наявності певних членів у класах-стратегіях. Хоча SFINAE — потужне правило метапрограмування на C++, у нього є один критичний недолік — помилки компілятора **дуже важко** зрозуміти. Тому, хоча воно дозволило Александреску побудувати гнучку систему у свій час, бібліотека Loki мала один критичний недолік: низьку читабельність коду та непрозорість повідомлень про помилки.

На щастя, у C++20 з'явився набагато простіший і чіткіший спосіб вказати обмеження на шаблонні аргументи — концепти. Концепти за своєю суттю — це іменований набір предикатів, яким повинен відповідати тип, що передається як параметр шаблону. Це безпосередньо корелює з ідеєю стратегій Александреску: в бібліотеці Loki вони були концептуальним, тобто неявно описаним, інтерфейсом. Насправді використання концептів несе за собою радикальну зміну парадигми: перехід від неявної перевірки під час підстановки параметрів шаблону до явної декларації вимог у вигляді інтерфейсу.

Взяти лише `SmartPtr` із Loki. У версії Александреску він поєднував різні стратегії (володіння, конвертація типів, перевірка і зберігання) через параметри шаблону. Тобто кожна стратегія в розумному вказівнику — це окремий

інтерфейс, якому має відповідати стратегія. Постає питання: де знайти цей інтерфейс, де він описаний? Він описаний у книзі Александреску, його можна спробувати вивести власноруч, читаючи сирцевий код. Але ніде чітко інтерфейс не занотований. Якщо спробувати підставити свою стратегію, яка не буде відповідати цьому *неявному* інтерфейсу, наприклад, ми забудемо реалізувати функцію `void OnDereference()` в стратегії перевірки, то отримаємо доволі дивне повідомлення, що функції бракує в якомусь `KP`; щоб дізнатись, що це стратегія перевірки, необхідно читати сирцевий код бібліотеки.

```
loki/Reference/SmartPtr.h: In instantiation of
 ↪  ‘Loki::SmartPtr<T, OwnershipPolicy,
 ↪  ConversionPolicy, CheckingPolicy,
 ↪  StoragePolicy>::ReferenceType Loki::SmartPtr<T,
 ↪  OwnershipPolicy, ConversionPolicy, CheckingPolicy,
 ↪  StoragePolicy>::operator*() [with T = int;
 ↪  OwnershipPolicy = Loki::DeepCopy; ConversionPolicy =
 ↪  Loki::DisallowConversion; CheckingPolicy = NoCheck;
 ↪  StoragePolicy = Loki::DefaultSPStorage;
 ↪  ReferenceType = int&]’:
main.cpp:28:19:   required from here
   28 |     std::cout << *ptr;
      |                  ^~~
loki/Reference/SmartPtr.h:916:30: error: ‘OnDereference’
 ↪  is not a member of ‘Loki::SmartPtr<int,
 ↪  Loki::DeepCopy, Loki::DisallowConversion,
 ↪  NoCheck>::KP’ {aka ‘NoCheck<int*>’}
  916 |           KP::OnDereference(GetImplRef(*this));
      |           ~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~
```

Натомість у Loki23 кожна стратегія, **кожен інтерфейс** описаний явно у вигляді концептів C++, наприклад стратегія перевірки вказівника під час розіменування явно декларує три необхідні функції:

```
template <typename Policy, typename Tptr>
concept CheckingPolicy = requires(Policy policy, Tptr ptr) {
  { policy.on_init(ptr) } -> std::same_as<void>;
  { policy.on_dereference(ptr) } -> std::same_as<void>;
```

```
    { policy.on_release(ptr) } -> std::same_as<void>;
  };
```

Лістинг 4: Концепт `CheckingPolicy` для явного опису стратегії перевірки у `Loki23`

- Функція `void on_init(Tptr)` слугує ініціалізатором
- Функція `void on_dereference(Tptr)` робить необхідні перевірки перед розіменуванням
- Функція `void on_release(Tptr)` виконується, коли розумний вказівник втрачає власність над ресурсом

Якщо користувач хоче надати реалізацію власної стратегії перевірки, йому треба виконати цей інтерфейс. Якщо ж його клас не відповідатиме цьому інтерфейсу, компілятор явно про це повідомить. Причому повідомить не тоді, коли, наприклад, відсутня функція буде використана, як у прикладі вище, а під час самої підстановки. Тобто концепти оберігають розробника від ситуації, коли інтерфейс *частково* виконаний, але частини, яких немає, просто не використовуються в коді і тому компілятор не повідомляє про помилку (тому що шаблони підставляються ліниво).

2. Використання constexpr та consteval для перенесення обчислень у стадію компіляції

Не менш цікавою зміною у C++11 стали обчислення під час компіляції. Взагалі, можливість обчислень під час компіляції була в C++ завжди, буквально від самого народження мови. Константні вирази у своєму найпростішому вигляді описані ще в K&R C: «A constant expression is an expression that involves only constants. Such expressions may be evaluated during compilation rather than run-time, and accordingly may be used in any place that a constant can occur» [7], там само наводиться декілька прикладів константних виразів:

```
#define MAXLINE 1000
char line[MAXLINE+1];

#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Лістинг 5: Приклади константних виразів у мові C

Також визначаються місця, де константні вирази можуть бути використані: ініціалізатор та розмірність масиву, мітки в операторі **switch** тощо.

Окрім константних виразів із C, у C++98 доступний ще один метод обчислень під час компіляції: метапрограмування мовою шаблонів є, по суті, чистим функціональним програмуванням, яке виконується під час компіляції, а готова програма містить лише результат. Але таке програмування обмежене, по-перше, системою типів у C++, наприклад, метапрограма не може оперувати рядками чи числами з плавучою крапкою, а по-друге, код, написаний функціональною мовою шаблонів, дуже сильно відрізняється від коду, написаного на ”звичайному” C++. Таким чином, хоча мова шаблонів і є повною за Тьюрингом підмножиною C++, вона була створена як реалізація узагальненого програмування, а обчислення під час компіляції — всього лише побічний ефект цієї реалізації. Тому обчислення під час компіляції за межами метапрограмування не були використані в Loki.

Проте ще з константними виразами в стилі C виникла проблема, яку Б’ярне Строуструп та інші члени комітету стандартизації C++ описали як «complex rules for simple things» [2]: до появи C++11 не було чіткого визначення, що ж таке константний вираз. У тому ж K&R, наприклад, він визначений як ”вираз, в який входять лише константи”, але не зрозуміло, чи маються на увазі логічні константи (як `constant_2`) чи фізичні `constant_1` нижче:

```
const int constant_1 = 5;
const int constant_2 = argc;
const int is_constant = constant_1 + constant_2;
```

Лістинг 6: Приклад залежності константного виразу від значення часу виконання

Як уже було сказано, у Modern C++ доступний новий спосіб організувати обчислення часу компіляції — *узагальнені* константні вирази. **constexpr** — це не окрема підмножина мови, а всього лише специфікатор, який говорить компілятору, що змінна є константним виразом або що функція *може бути* обчислена під час компіляції. У C++11 це був принципово новий спосіб перенесення обчислень на час компіляції. Але перехід до обчислень часу компіляції не був раптовим: у C++11 були накладені жорсткі вимоги до

функцій, які можуть бути обчислені під час компіляції; наприклад, функція могла використовувати лише один оператор **`return`**. Це спонукало писати доволі дивний код, використовуючи тернарні оператори (адже повертати значення в двох гілках **`if`**/**`else`** не можна було). Ці обмеження були зроблені навмисно: якби члени комітету стандартизації C++ від самого початку дозволили використовувати, скажімо, повну функціональність C++ (як мови, так і стандартної бібліотеки), жоден компілятор би не встиг це підтримати до виходу стандарту. Натомість вимоги до функцій, що можуть бути обчислені під час компіляції, з кожним стандартом ставали ліберальнішими, і вже починаючи з C++20 у **`constexpr`** функціях можна використовувати *майже весь* C++. Фактично константні вирази із ”ініціалізатора масиву” перетворились на одну з найпотужніших особливостей C++.

Окрім обчислень часу компіляції ключове слово **`constexpr`** отримало ще одне **дуже незвичне** використання в умовних операторах часу компіляції **`if constexpr`**. Інколи умовний перехід необхідно зробити під час компіляції. Раніше для цього необхідно було використовувати мову шаблонів разом із SFINAE, проте якщо SFINAE вже використовується, **`if constexpr`** дозволяє реалізувати умовні переходи *не збільшуючи* кількість перевантажень шаблонних класів.

Зрештою, у C++20 було додано ключове слово **`consteval`**, яке **вимагає** обчислити функцію під час компіляції. Річ у тому, що **`constexpr`** всього лише каже, що функція *може бути* обчислена під час компіляції, але не зобов’язує компілятор це робити. Це створює доволі явне неузгодження очікувань розробників і стандартної поведінки, яке **`consteval`** виправляє.

3. Варіативні шаблони та Fold Expressions у сучасних стратегіях

Серцем бібліотеки Loki були, безумовно, списки типів: вони були всюди, від фабрик об’єктів до функторів. Але вони мали фундаментальне обмеження через власну конструкцію: список будується як рекурсивна вкладена структура `Typelist<Head, Tail>`, де `Tail` або є іншим `Typelist`, або `NullType`. Це так звані cons-cell об’єкти, запозичені з LISP. Для спрощення синтаксису Loki визначає макроси `TYPELIST_1 ... TYPELIST_50`, що перетворюють рекурсію у плоский список:

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
```

```
#define TYPELIST_2(T1,T2) Typelist<T1, TYPELIST_1(T2)>
#define TYPELIST_3(T1,T2,T3) Typelist<T1, TYPELIST_2(T2,T3)>
// ... till TYPELIST_50

// usage:
typedef TYPELIST_3(int, double, char) MyTypes;
```

Лістинг 7: Макроси `TYPELIST_1`, `TYPELIST_2` і `TYPELIST_3` у класичній `Loki`

Макроси, звичайно, спрощують використання списку типів, але обмежують кількість цих типів до 50 (або змушують самостійно додавати макроси), ускладнюють налагодження (препроцесор приховує справжню структуру коду) і є несумісними з ідіомою передачі параметрів, оскільки програміст пише макрос, а справжній тип із рекурсивною структурою розгортається компілятором під капотом, неявно.

На щастя, у C++11 були стандартизовані варіативні шаблони. Їхня варіативність полягає в тому, що кількість шаблонних параметрів не фіксована, як було раніше, а може *варіюватись* від виклику до виклику, дозволяючи шаблону приймати довільну кількість параметрів. Насправді варіативність у кількості аргументів функції була ще з часів C за допомогою *еліпсису*, власне, так і працює `printf()`:

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

void dummy_printf(const char *format, ...)
{
  size_t count = strlen(format) / 2;

  va_list list;
  va_start(list, count);

  for (size_t arg{}; arg != count; ++arg) {
    if(strncmp(format + 2 * arg, "%d", 2) == 0)
      std::cout << "[int]: " << va_arg(list, int) << '\n';
    else if(strncmp(format + 2 * arg, "%s", 2) == 0)
      std::cout << "[str]: " << va_arg(list, const char*) << '\n';
  }

```

```
  va_end(list);
}

int main()
{
  dummy_printf("%d", 1);                // 1
  dummy_printf("%s", "foo");           // foo
  dummy_printf("%d%s%d", 3, "bar", 5); // 3 bar 5
}
```

Лістинг 8: Реалізація варіативної функції `dummy_printf` через еліпсис

Але такий підхід, як і більшість відносно зручних конструкцій на C, не є типобезпечним і породжує помилки, які мали б перевірятися, але не перевіряються під час компіляції. Однією з причин додавання варіативних шаблонів у C++ є проблема типонебезпечного `printf()` «[variadic template] Enables the implementation of a type-safe, C++-friendly printf() ...» [5].

Стандарт C++17 доповнив варіативні шаблони виразами згортання (fold expressions), які дозволяють застосовувати бінарні оператори до всіх типів у наборі. Особливо це показово в модернізованому варіанті `GenScatterHierarchy` — патерні Loki, що генерує клас, успадкований від `Unit<T>` для кожного `T` у списку. Оригінальна реалізація спирається на рекурсивне успадкування:

```
template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<Typelist<T1, T2>, Unit>
    : public GenScatterHierarchy<T1, Unit>
    , public GenScatterHierarchy<T2, Unit>
{
  // ...
}
```

Лістинг 9: Рекурсивна реалізація `GenScatterHierarchy` у класичній `Loki`

Натомість у Loki23 для цього були використані оператори згортки, що не лише дозволило скоротити запис майже вдвічі:

```
template <typename... Ts, template <typename> class Unit>
struct GenScatterHierarchy<TypeList<Ts...>, Unit>
    : public Unit<Ts>...
```

```
{
  // ...
}
```

Лістинг 10: Реалізація `GenScatterHierarchy` у `Loki23` з використанням варіативних шаблонів

Варіативні шаблони роблять використання списку типів набагато прозорішим: у прикладі вище успадкування із `...` одразу повідомляє програмісту, що клас успадковує `Unit<T>` для кожного типу з аргументів без необхідності прочитання трьох спеціалізацій і уявного ”розгортання” рекурсії, як це було в Александреску.

Врешті-решт, варіативні шаблони — це **стандартний** механізм декларації змінної кількості шаблонних параметрів. Їх використання означає, що, по-перше, недосвідченим розробникам буде легше розібратися з ними, а досвідчені зможуть застосовувати знання варіативних шаблонів до списків типів.

Список типів — це, напевно, частина Loki, яка застаріла найбільше, але не тому, що Андрей Александреску приділив їй замало уваги, а тому, що спосіб реалізації і, власне, мова, якою вона написана, застаріли ще з виходом 11-го стандарту.

4. Переваги модульної системи для архітектури бібліотек шаблонів

Бібліотека Loki, як і більшість бібліотек того часу, побудована у форматі заголовних файлів. Це є стандартною практикою для шаблонних бібліотек C++ навіть зараз, оскільки шаблони інстанціюються компілятором у кожній одиниці трансляції, їхні визначення мають бути видимі в момент компіляції. Проте такий підхід спричиняє декілька добре відомих проблем. По-перше, заголовні файли не мають жодного механізму ізоляції: усі макроси, імпортування простору імен чи статичні допоміжні сутності, оголошені в заголовку, потрапляють до *кожної* одиниці трансляції, що включила його. Це є джерелом конфліктів імен через *забруднення простору імен*. По-друге, заголовкові файли обробляються препроцесором і компілятором повторно при кожному включенні. Тобто якщо є умовний великий заголовний файл і його включають в `N` файлів, однаковий код буде проходити лексичний, синтаксичний і

семантичний аналізи `N` разів. Таким чином час компіляції великих проектів, що активно використовують Loki (чи будь-яку іншу бібліотеку, яка складається здебільшого із заголовних файлів), збільшується через повторну обробку заголовків.

Щоб розв’язати ці проблеми, у стандарті C++20 введено нову систему розділення одиниць трансляції на *модулі*. На відміну від заголовних файлів, у яких усе, що було оголошено, імпортувалося тим, хто цей файл включає, модуль приватний за замовчуванням: щоб модуль щось експортував, треба це вказати явно. Окрім цього, включення модуля — це не проста текстова підстановка. Натомість модуль компілюється в проміжний формат, який надалі зчитується компілятором, якщо цей модуль було включено в інший файл. У 2023 році компанія Alibaba навіть провела дослідження, наскільки перехід на модулі пришвидшує компіляцію великої шаблонної кодової бази (понад 500 тис. рядків). Виявилось, що перехід на модульну архітектуру для їхнього проекту скоротив загальний час збирання на 42% [8].

РОЗДІЛ 3. ВПЛИВ LOKI НА MODERN C++

Було б несправедливо розглядати Loki Александреску як ще одну застарілу бібліотеку, яка цінна лише в академічній сфері. Опублікувавши книгу ”Modern C++ Design” Андрей Александреску продемонстрував, що поєднання шаблонів, множинного успадкування та часткової спеціалізації дозволяє перетворити C++ з ”об’єктно-орієнтованого” C на мову програмування із повноцінною системою метапрограмування на етапі компіляції для побудови абстракцій. Бібліотека Loki стала втіленням цієї ідеї, у ній Александреску запропонував компоненти, які пізніше лягли в основу стандартів C++11, C++14, C++17 та C++20.

1. Бібліотека метапрограмування та перевірки часу компіляції в Modern C++

До виходу стандарту C++11 стандартна бібліотека шаблонів (STL) взагалі нічого не знала про метапрограмування. У C++03 шаблони — це лише метод узагальненого програмування, який використовувався для реалізації контейнерів чи узагальнених функцій.

Натомість Андрей Александреску в Loki запропонував інший погляд на мову шаблонів: замість звичайного узагальненого програмування він використав її для реалізації метапрограмування і своєрідної рефлексії в C++. Такий підхід виявився надзвичайно зручним, адже інколи поведінку функції чи класу необхідно скоригувати залежно від *типу* шаблонного аргумента. Візьмемо за приклад функцію, яка друкує дані типу `T` на екран, якщо `T` — це число чи символ, ми його друкуємо одним чином, але якщо це вказівник, то можемо його розіменувати й надрукувати те, на що він вказує. Саме тому багато базових шаблонів, які виконували інтроспекцію типів у Loki, були реалізовані в окремій бібліотеці шаблонів `<type_traits>` в C++11 і наступних:

- `Int2Type<i>` → `std::integral_constant<int, i>`
- `Type2Type<T>` → `std::type_identity<T>`
- `Select<b, T, E>::Result` → `std::conditional_t<b, T, E>`
- `Conversion<T, E>::exists` → `std::is_convertible_v<T, E>`

Властивості типів у Loki — це один із небагатьох випадків, коли бібліотека стала настільки зручною і корисною, що частину її функціональності реалізували прямо в стандартній бібліотеці мови.

Не менш цікавим прикладом запозичення стандартною бібліотекою шаблонів функціональності Loki є перевірка часу компіляції. Річ у тому, що до C++11 не існувало явної можливості перевірити *якусь умову*, яка відома під час компіляції, і видати помилку, якщо вона не справдилася. Натомість ще з часів C існувала директива препроцесора `#error`, яка використовувалась разом з іншими макросами препроцесора, наприклад для перевірки версії компілятора:

```
#if __GNUC__ < 5
    #error "This code requires GCC version 5 or higher."
#endif
```

Лістинг 11: Директива препроцесора `#error` для перевірки версії компілятора

Loki запропонувала інший підхід: додати можливість генерувати помилки не препроцесором, а всередині самого C++. Звичайно, в таких перевірках можна використовувати лише інформацію, яка відома під час компіляції, наприклад, інформацію про тип з бібліотеки метапрограмування. Результатом став макрос `STATIC_CHECK(expr, msg)`, який створював локальний клас, назва якого містила `msg`, якщо вираз був хибним. Це спричиняло помилку компіляції, де в повідомленні фігурувало ім'я цього класу, що допомагало розробнику зрозуміти причину помилки. У Modern C++ це реалізовано на рівні мови: у C++11 було додано ключове слово `static_assert()`, яке дозволяє передавати умову і рядок з поясненням безпосередньо.

2. Функтори Modern C++ як нащадки Loki

Не менш важливою частиною Loki є функтори — універсальна *узагальнена* обгортка, яка може інкапсулювати будь-яку сутність, що підтримує звернення за оператором виклику (круглих дужок). При цьому такі функтори зберігали строгу типізацію аргументів і типу повернення. Проблема, яку вирішував Александреску, була фундаментальною для C++: у C++98 існувало щонайменше чотири різні типи сутностей, які можна викликати:

- звичайні функції

- вказівники на функції
- вказівники на функції-члени
- власне функтори (об’єкти з перевантаженим `operator()`)

Але не існувало жодного єдиного механізму, який міг би зберігати та *уніфіковано викликати* будь-яку з них. Окрім того, стандартні методи карінгу аргументів за допомогою `bind1st` чи `bind2nd` обмежувалися одним аргументом і не підтримували вказівники на функції-члени, а `std::pointer_to_unary_function()` та споріднені адаптери вимагали ручного обирання конкретного класу для кожного випадку.

`Functor<T, TypeList>` в Loki дозволяє розв’язати цю проблему. Він зберігає об’єкт, що викликається, типи його аргументів та тип значення, що повертається. Такий клас повністю підтримує семантику копіювання, присвоєння та передачу за значенням (наприклад, у функцію, яка хоче його модифікувати). Він не є поліморфним і не призначений для успадкування, а тому не вимагає створення віртуальної таблиці функцій. Вартість використання становить один рівень непрямоти для простих об’єктів, плюс додатковий віртуальний виклик для кожного зв’язаного параметра.

Окрім базового зберігання та виклику, `Functor` в Loki реалізував дві потужні композиційні операції. Перша — `BindFirst` — дозволяє зафіксувати перший аргумент функтора, створюючи новий функтор з меншою арністю. У функціональних мовах програмування це називається карінг. Наприклад, маючи `Functor<void, TYPELIST_2(int, int)>`, можна зафіксувати перший аргумент значенням 10 і отримати `Functor<void, TYPELIST_1(int)>`. Прив’язування може застосовуватися повторно, поступово зменшуючи кількість необхідних аргументів до нуля. Александреску описував прив’язування як механізм, що дозволяє зберігати разом із логікою обчислення частину його параметрів, що дає змогу поступово зменшити кількість інформації, необхідної під час виклику.

Друга операція — ”операція ланцюжка”, або композиція функцій у функціональному програмуванні. Вона дозволяє об’єднати кілька функторів в один, що при виклику послідовно виконує всі вкладені виклики. Для цього реалізований клас `FunctorChain`, який зберігає два функтори, а його `operator()` викликає їх послідовно. Важливо, що ні реалізація карінгу,

ані композиції не вимагали жодних змін у самому Functor, що доводило ортогональність дизайну.

Шлях до стандартизації такого механізму був на диво складним. По-перше, в функторів була одна **критична** проблема, яка офіційно називається ”The Forwarding Problem” (”Проблема перенаправлення”): до C++11 було неможливо написати таку функцію, яка ефективно (тобто без створення зайвих копій) перенаправляла аргументи в іншу функцію. Loki довелося балансувати між копіюванням аргументів і використанням посилань, але це не було універсальним рішенням. Можливість ефективної передачі аргументів з’явилася лише в 2002 році, коли була опублікована пропозиція, яка реалізувала новий синтаксис `T&&` [1].

Врешті, в 2002 році була опублікована пропозиція N1402, яка реалізувала універсальні функтори в STL [4].

3. Еволюція розумних вказівників у C++

Розумні вказівники є, мабуть, найпопулярнішою, найскладнішою та найпотужнішою ідіомою C++. Як зазначає Александреску на початку ”Modern C++ Design”, розумні вказівники поєднують безліч синтаксичних та семантичних питань: вони імітують звичайні вказівники через `operator→()` та `operator*()`, але додають автоматичне управління часом життя об’єкта. Ключове спостереження полягає в тому, що хоча розумний вказівник є значенням (власне, вказівником), він бере на себе відповідальність за знищення об’єкта, на який вказує.

Фундаментальна складність полягає в тому, що не існує єдиної універсальної стратегії управління володінням. Різні сценарії вимагають різних підходів: глибоке копіювання для поліморфних об’єктів, підрахунок посилань для спільного володіння, зв’язування посилань для середовищ із обмеженою купою та деструктивне копіювання для передачі виключного володіння. Кожна стратегія має свої компроміси щодо витрат пам’яті, швидкодії, потокобезпечності чи семантичної коректності в конкретному контексті використання. До появи Loki стандарт C++98 пропонував лише один розумний вказівник — `std::auto_ptr`, що реалізовував стратегію деструктивного копіювання. Але він мав один фундаментальний дефект: його конструктор копіювання та оператор присвоєння приймали неконстантне посилання,

порушуючи конвенції C++ і роблячи його несумісним із контейнерами STL. Передача такого вказівника за значенням поглинала оригінальний вказівник: після виклику `f(auto_ptr)` оригінальний `auto_ptr` буде містити нульовий вказівник. Це не те, чого очікував би розробник, і така поведінка легко створює дивні помилки із розіменуванням вказівника на `NULL`. Саме тому в Loki клас розумного вказівника був реалізований за принципом стратегій, із можливістю його налаштування під конкретні потреби.

Стандартна бібліотека C++11 обрала принципово інший шлях, ніж пропонувала Loki: замість одного параметризованого `SmartPtr` із комбінаторною гнучкістю — два чітко розділені типи, кожен із яких обслуговує одну стратегію володіння. `std::unique_ptr<T, Deleter>` реалізує виключне володіння — рівно один вказівник володіє об’єктом у будь-який момент часу. Це пряма заміна `DestructiveCopy` з Loki, але виправлена завдяки rvalue-посиланням: замість мовчазного обнулення джерела при копіюванні, `std::unique_ptr<T, Deleter>` забороняє копіювання взагалі та дозволяє лише переміщення. Спроба скопіювати цей вказівник — це помилка компіляції, а не тиха поведінкова пастка. Параметр `Deleter` є статичною стратегією — прямий спадок дизайну на основі стратегій з Loki. За замовчуванням `std::default_delete<T>` викликає `delete`. Користувач може надати власний `Deleter`: функтор або вказівник на функцію для закриття файлів, звільнення пам’яті, повернення об’єкта до пулу або будь-якої іншої логіки очищення. `std::shared_ptr<T>` реалізує спільне володіння через підрахунок посилань; по суті, це аналог `Loki::SmartPtr<T, RefCounted>`.

РОЗДІЛ 4. АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ БІБЛІОТЕКИ LOKI23

Оригінальна бібліотека Loki, описана у книзі Александреску, безумовно, здійснила справжню революцію у світі програмування на C++. Проте варто зазначити, що, оскільки вона створювалася для стандарту C++98, її внутрішня реалізація містила значну кількість архітектурних компромісів, обхідних шляхів, надзвичайно складних рекурсивних шаблонних конструкцій та небезпечних і незручних макросів. Це було зумовлено серйозними обмеженнями тогочасних компіляторів та самої мови, яка ще не мала вбудованих механізмів для розв'язання багатьох завдань узагальненого програмування, на передовій якого тоді стояв Андрей Александреску. З виходом нових стандартів, починаючи з C++11 і закінчуючи C++23, мова C++ зазнала кардинальних змін. Фактично тепер ми говоримо про зовсім іншу мову, називаючи її "Modern C++". З'явилися варіативні шаблони, які назавжди змінили підхід до роботи зі списками типів, семантика переміщення, що підвищила продуктивність програм шляхом уникнення зайвих копіювань, вирази згортки шаблонів, які дозволили писати лаконічний код замість громіздкої рекурсії, строга та передбачувана модель пам'яті для багатопотоковості, концепти, модулі тощо. Впровадження всіх цих конструкцій дозволило не лише значно підвищити продуктивність коду та типобезпеку, а й суттєво спростити внутрішню реалізацію складних метапрограм, зробивши їх зрозумілішими для ширшого кола розробників.

Бібліотека Loki23 є прямим результатом цього масштабного переосмислення: вона зберігає концептуальну гнучкість та елегантність дизайну на основі стратегій, за який цінується Loki, але повністю переписує її фундамент інструментами сучасного C++, перетворюючи академічний експеримент на надійний, швидкий та безпечний інструмент для розробки сучасних програмних систем.

1. Переосмислення ієрархії стратегій за допомогою обмежень та концептів

Як уже було неодноразово сказано, дизайн на основі стратегій є ключовою *архітектурною* парадигмою класичної бібліотеки Loki. Перевага такого підходу полягає в тому, що він дозволяє створювати надзвичайно гнучкі

та легко розширювані класи шляхом комбінування простих, сфокусованих на одному завданні та ортогональних компонентів — стратегій. Клас-хост збирає ці стратегії, отримуючи потрібний набір поведінок, який конфігурується розробником виключно на етапі компіляції. Це усуває необхідність у використанні віртуальних функцій та пов'язаних із ними накладних витрат часу виконання, зберігаючи при цьому поліморфну поведінку. Однак у стандартах до C++20 існувала серйозна проблема: перевірка того, чи дійсно переданий тип стратегії задовольняє всім необхідним синтаксичним та семантичним вимогам (наприклад, чи має він потрібні методи з правильними сигнатурами, чи підтримує певні оператори), відбувалася лише на етапі інстанціювання тіла самого шаблону. Цей підхід називається неявною або "качиною" типізацією. Якщо користувач через помилку передавав некоректний тип, який не мав потрібного методу, компілятор намагався розгорнути шаблон, заглиблюючись у його внутрішні виклики та рекурсії. Зрештою, натрапивши на відсутній метод або несумісний тип десь глибоко в надрах бібліотеки, відбувалася фатальна помилка компіляції. Як наслідок, компілятор видавав величезні, абсолютно незрозумілі повідомлення про помилки, які часто сягали сотень, а то й тисяч рядків нечитабельного тексту для однієї-єдиної помилки. Це робило використання шаблонних бібліотек серйозною проблемою для новачків. В епоху C++11/14/17 цю проблему намагалися частково вирішувати за допомогою складної і заплутаної ідіоми SFINAE або використання стандартних метафункцій типу `std::enable_if`. Проте такий підхід можна було назвати радше тимчасовим обхідним рішенням, ніж повноцінним розв'язанням, адже він усе одно був важким для розуміння та підтримки. У бібліотеці Loki23 цей підхід був фундаментально переосмислений шляхом використання концептів та обмежень, які були представлені у стандарті C++20 як одна з найважливіших його особливостей. Концепти дозволяють розробнику бібліотеки явно, декларативно специфікувати синтаксичні та семантичні вимоги до параметрів шаблону безпосередньо у його сигнатурі, створюючи строгий контракт між бібліотекою та користувачем. Впровадження концептів у архітектуру Loki23 має декілька надзвичайно вагомих переваг, які суттєво покращують якість коду, який її використовує:

1. Вимоги до стратегій тепер є формалізованою і невіддільною частиною відкритого інтерфейсу класу. Будь-який розробник, дивлячись на

оголошення шаблону, одразу чітко бачить, що саме вимагається від стратегії

2. Якщо переданий користувачем тип не задовольняє вказаний концепт, компілятор зупиняє роботу і видає коротке, точне та зрозуміле повідомлення про помилку (наприклад, ”тип X не задовольняє концепту Y, оскільки не має методу Z”) ще до того, як спробує інстанціювати складне тіло шаблону

3. Концепти можуть використовуватися для вибору найбільш слушної спеціалізації шаблону залежно від властивостей типу, повністю замінюючи собою великі конструкції SFINAE. Компілятор тепер може алгоритмічно визначати, який концепт є більш специфічним, обираючи найкращий варіант

4. Компілятору більше не потрібно марно витрачати ресурси та час на розгортання помилкових гілок шаблонів до самого кінця. Він просто відкидає типи, які не проходять перевірку концептом, на самому початку процесу підстановки шаблонних аргументів

Для наочного прикладу того, як це реалізовано, розглянемо модуль `loki23::hierarchy`. У ньому впроваджено концепт `Bindable`, який формально вимагає наявності шаблонного методу `bind` із заданою сигнатурою:

```
template <typename T, typename E, typename... Args>
concept Bindable = requires(T t) {
  { t.template bind<Args...>(std::declval<Args>()...) }
    -> std::same_as<void>;
};
```

Лістинг 12: Концепт `Bindable` для формалізації вимог до шаблонного методу `bind`

Цей концепт застосовує ключове слово `requires` для створення так званого виразу вимог. Усередині цього виразу компілятор перевіряє, чи є синтаксично коректним виклик шаблонного методу `bind()` на фіктивному об’єкті `t` типу `T` з аргументами типу `Args...` (використовуючи `std::declval` для уникнення необхідності створення реальних об’єктів). Крім того, за допомогою

стрілочного синтаксису → `std::same_as<void>` концепт жорстко перевіряє, чи повертає цей метод саме тип `void`, а не щось інше.

Такий формалізований підхід є одним із найважливіших архітектурних аспектів Loki23: кожна вимога, з якою може взаємодіяти користувач, має бути декларативно описана. Наприклад, якщо користувач захоче розширити розумний вказівник, реалізувавши власну стратегію підрахунку посилань, він повинен буде створити клас, який задовольняє відповідний концепт. Вимоги до цього класу не будуть розкидані *десь* по класу розумного вказівника, натомість вони декларативно описані в самому концепті стратегії:

```
template <typename Policy, typename Thandle>
concept OwnershipPolicy = requires(Policy policy, Thandle h) {
  { policy.acquire(h) } -> std::same_as<void>;
  { policy.release(h) } -> std::same_as<bool>;
};
```

Лістинг 13: Концепт `OwnershipPolicy` для формалізації стратегії володіння ресурсом

2. Реалізація сучасного розумного вказівника з підтримкою атомарних операцій та власних стратегій володіння

Класичний `Loki::SmartPtr` був неймовірно інноваційним інструментом в епоху до появи стандартизованих та безпечних розумних вказівників, таких як `std::unique_ptr` та `std::shared_ptr`, які були додані лише у стандарті C++11. До того часу розробники мали у своєму арсеналі лише сумнозвісний `std::auto_ptr`, використання якого часто призводило до витоків пам'яті або невизначеної поведінки через його дивну семантику деструктивного копіювання. Однак навіть у сучасному C++, попри величезну популярність стандартних розумних вказівників, вони все одно не завжди пропонують розробникам достатню гнучкість для вирішення специфічних та нестандартних архітектурних завдань. Стандартна бібліотека C++ пропонує універсальні рішення для 95% випадків, але для решти 5% складних систем потрібні спеціалізовані інструменти. Можна виділити такі суттєві обмеження стандартних розумних вказівників:

- `std::shared_ptr` завжди і безумовно виділяє так званий контрольний блок для зберігання лічильника посилань на купі. У системах реального

часу, іграх, ядрах операційних систем або вбудованих системах з обмеженими ресурсами часті алокації пам'яті на купі є категорично неприйнятними через фрагментацію та непередбачувані затримки.

- У стандартних вказівниках повністю відсутня можливість легко та декларативно змінити поведінку програми при перевірці вказівника перед розіменуванням. Наприклад, якщо програма намагається розіменувати нульовий `std::shared_ptr`, відбудеться аварійне завершення програми. І хоча це вкладається в ідеологію відсутності накладних витрат у C++, на жаль, це також призводить до нескінченної кількості помилок у коді, адже розробник не може налаштувати вказівник так, щоб замість аварійного завершення він, наприклад, записав повідомлення про помилку до журналу, спровокував виняток та дозволив програмі продовжити роботу або ініціював процедуру м'якого відновлення стану системи.

- Стандартні розумні вказівники розроблялися переважно для керування сирими вказівниками на пам'ять. Їх досить складно та незручно використовувати для створення безпечних RAII-обгорток над системними ресурсами, що не є власне пам'яттю.

Loki23 намагається розв'язати ці проблеми завдяки ідеї проектування на основі стратегій Александреску: замість використання фіксованого набору поведінок, жорстко закріплених за конкретними класами розумних вказівників, `Loki23::SmartPtr` є узагальненим контейнером, який користувач може налаштувати під конкретні потреби для управління будь-якими ресурсами. Гнучкість конфігурації досягається внаслідок використання незалежних стратегій.

```
template <typename Tval,
          typename OwnPolicy   = DeepCopy,
          typename CheckPolicy = NoCheck,
          template <typename> typename StorePolicy =
           ↪  DefaultStorage,
          typename ConvPolicy  = DisallowConversion>
  requires ((!std::same_as<OwnPolicy, IntrusiveOwnership> ||
             (IntrusiveCounted<Tval> &&
```

```
            std::same_as<typename
              ↪  StorePolicy<Tval>::handle_type, Tval *>) &&
          ConversionPolicy<ConvPolicy, typename
           ↪  StorePolicy<Tval>::handle_type>)
class SmartPtr;
```

Лістинг 14: Оголошення `SmartPtr` у `Loki23` із параметризованими стратегіями

По-перше, наведений вище фрагмент коду демонструє виразність та потужність концептів C++20. За допомогою ключового слова `requires` безпосередньо в оголошенні класу накладається складне логічне і декларативне обмеження на шаблонні аргументи. По-друге, цей лістинг коду демонструє **надзвичайну** конфігурованість класу розумного вказівника: бібліотека визначає три осі конфігурації:

- `OwnPolicy` — стратегія копіювання та підрахунку посилань: хто є власником ресурсу та що відбувається при копіюванні вказівника. Наприклад, якщо користувач хоче використати підрахунок посилань, він може використати `RefCounted`, або `NoCopy`, якщо треба заборонити копіювання об'єктів вказівника.
- Стратегія `CheckPolicy` дозволяє конфігурувати поведінку при зверненні через недійсний (нульовий) вказівник. Користувач може вибрати `NoCheck` для відсутності перевірки або `ExceptionCheck`, щоб кидати виняток при спробі розіменування нульового вказівника тощо.
- `StorePolicy` визначає дескриптор та спосіб звільнення ресурсу: вказівник, файловий дескриптор, сокет тощо.
- `ConvPolicy` визначає дозвіл або заборону неявного перетворення `SmartPtr` до сирого дескриптора. Це зручно при роботі із старою кодовою базою, яка очікує сирі вказівники, але при цьому дозволяє зберегти безпеку та контроль над ресурсами в новому коді.

3. Модернізація патерну Singleton: потокобезпека та життєвий цикл об'єктів у C++23

Патерн Singleton, вперше описаний "Бандою Чотирьох" [3], є, мабуть, одним із найвідоміших, найпоширеніших і найбільш жорстко критикованих

патернів у спільноті C++ та серед розробників загалом. Багато хто навіть відносить його до категорії ”антипатернів” (anti-patterns) через те, що він приховує глобальний стан та ускладнює модульне тестування. Проте у багатьох системних завданнях його використання залишається виправданим. Основними проблемами класичних реалізацій цього патерну в C++ завжди були дві:

1. Найпростіша, наївна реалізація відкладеної ініціалізації не є потокобезпечною у багатопотоковому середовищі: кілька потоків можуть одночасно побачити нульовий вказівник і створити кілька екземплярів одного об’єкта Singleton, спричиняючи витік пам’яті. Натомість використання `std::mutex` при кожному доступі до екземпляра негативно впливає на продуктивність такої реалізації патерну. Спроба обійти цю проблему продуктивності через винайдення патерну ”Подвійна перевірка” (Double-Checked Locking [10]) у стандартах до C++11 була принципово помилковою та небезпечною. Оскільки мова не мала формалізованої моделі пам’яті, агресивний компілятор під час оптимізації міг довільно переупорядковувати машинні інструкції читання та запису пам’яті. Це могло призвести до ситуації, коли один потік бачив ненульовий вказівник на виділену пам’ять, але конструктор об’єкта ще не встиг завершити роботу в іншому потоці, що призводило до використання неповністю ініціалізованого стану і краху програми. Добре, якщо програма одразу аварійно завершувалася, але вона могла і продовжити виконання в такому невизначеному стані.

2. Друга проблема полягає в непередбачуваному порядку знищення статичних об’єктів. Стандарт C++ не дає жодних гарантій щодо точного порядку ініціалізації та руйнування глобальних і статичних об’єктів, які знаходяться у різних, незалежних одиницях трансляції. Якщо один складний синглтон критично залежить від іншого синглтона під час завершення програми, виникає величезна небезпека звернення до вже знищеного об’єкта, і програма може успішно відпрацювати основний сценарій, але аварійно завершитися наприкінці, не закривши або не звільнивши якісь ресурси.

   У бібліотеці Loki23 модуль Singleton був повністю модернізований

з використанням сучасного C++ і проектування на основі стратегій Александреску. Гнучкість класу Singleton досягається завдяки тому, що він налаштовується користувачем через дві незалежні стратегії:

- Стратегія створення відповідає за те, як об'єкт фізично створюється в пам'яті та як він потім знищується. Наприклад, стратегія `CreateUsingNew` є найпростішою і використовує стандартний глобальний оператор new. Стратегія `CreateUsingMalloc` демонструє інтеграцію зі стилем програмування на C, використовуючи низькорівневі функції `std::malloc`, `std::free` та так званий placement new для виклику конструктора у виділеній пам'яті. Особливо цікавою є стратегія `CreateStatic`: вона гарантує, що об'єкт ніколи не буде виділятися на повільній купі. Замість цього вона використовує заздалегідь виділений масив статичної пам'яті правильного розміру та вирівнювання і конструює об'єкт прямо там.

- Стратегія тривалості життя бере на себе відповідальність керувати руйнуванням об'єкта під час завершення програми та визначати конкретну поведінку системи у критичному випадку виникнення проблеми звернення до Singleton після його знищення.

Якщо стратегії відповідають за гнучкість Singleton, то Modern C++ забезпечує його безпеку: реалізація доступу до єдиного екземпляра використовує високоефективний патерн Double-Checked Locking [10]. У Loki23 використовуються `std::atomic` та явна вказівка порядку доступу до пам'яті для уникнення будь-яких неузгодженостей упорядкування:

```
static T &instance() {
  T *p = pInstance_.load(std::memory_order_acquire);
  if (!p) {
    std::lock_guard lock{mutex_};
    p = pInstance_.load(std::memory_order_relaxed);
    if (!p) {
      if (destroyed_) {
        LifetimePol::on_dead_reference();
        destroyed_ = false;
      }
      p = CreationPol::create();
      pInstance_.store(p, std::memory_order_release);
```

```
        LifetimePol::schedule_destruction(p, &destroy_singleton);
      }
    }
    return *p;
}
```

Лістинг 15: Потокобезпечне отримання екземпляра `Singleton` через Double-Checked Locking

1. Перша швидка (тому що не використовує м'ютекс) перевірка використовує атомарне завантаження `std::memory_order_acquire`. Це встановлює так звану гарантію набуття ресурсу. Вона створює жорсткий односторонній бар'єр пам'яті, який суворо забороняє компілятору переносити будь-які інструкції читання пам'яті, що знаходяться в коді після цієї перевірки, вище цього виклику. Тому, якщо наш потік побачить ненульовий вказівник `p`, ми гарантовано бачимо повністю ініціалізований об'єкт у пам'яті, без жодного ризику натрапити на частково сконструйований стан.

2. Якщо вказівник виявився нульовим, потік змушений використати глобальне блокування об'єкта (той самий м'ютекс). Це доволі дорога системна операція, але найважливіше те, що вона відбувається лише під час першого звернення до Singleton.

3. Після успішного створення об'єкта через обрану стратегію `CreationPol::create()`, новий валідний вказівник записується назад у статичну змінну за допомогою `std::memory_order_release`. Це встановлює гарантію вивільнення. Вона використовує механізм когерентності кешу процесора і гарантує, що всі записи в пам'ять стануть видимими для всіх інших потоків.

4. Реалізація ієрархій класів

Одна з найбільш незвичних частин оригінальної бібліотеки Loki — це генерація ієрархій класів. За її допомогою бібліотека автоматично створює складні розгалужені або лінійні структури спадкування класів на основі простого списку переданих типів прямо на етапі компіляції. Завдяки цьому можна реалізувати патерн абстрактної фабрики декларативно.

В оригінальному, класичному Loki списки типів реалізовувалися як примітивні рекурсивні структури-вузли, а маніпуляції з ними вимагали написання величезної кількості допоміжного шаблонного коду для їх обходу, зшивання та пошуку. У Loki23 використання варіативних шаблонів дозволило відмовитися від подібних конструкцій, замінивши їх на значно простіший та ефективніший для компілятора контейнер `template <typename... Ts> struct type_list`. Бібліотека реалізує два принципово різні типи ієрархій: розгалужену та лінійну.

Розсіяна ієрархія базується на можливості множинного спадкування в C++: вона приймає як параметри базовий шаблон одиниці обробки, що інкапсулює логіку обробки типу, і варіативний список цільових типів. Підсумковий згенерований клас успадковує всі інстанціації базового шаблону-обробки для кожного типу, що входить до списку. Доступ до потрібного ”підкомпонента” (тобто конкретного базового класу зі списку) всередині такої розсіяної ієрархії здійснюється за допомогою вільної допоміжної функції. Така ієрархія дозволяє побудувати патерн абстрактної фабрики, де кожен клас зберігається в самій ієрархії.

На відміну від розсіяної, лінійна ієрархія будує послідовний вертикальний ланцюжок спадкування: обробник для типу T1 публічно спадкує обробник для типу T2, який, у свою чергу, спадкує обробник для T3, і так далі до кінця списку. Ця послідовність типів необхідна для імплементації, наприклад, систем диспетчеризації, де строгий порядок перевірки чи обробки має важливе значення.

5. Мультиметоди в Modern C++

Класичний механізм віртуальних функцій C++ за своєю природою забезпечує лише одинарну диспетчеризацію: поліморфна поведінка залежить лише від динамічного типу одного об’єкта, тобто того, через який безпосередньо здійснюється виклик методу. Однак у складних системах виникає необхідність виконувати специфічну логіку на основі динамічних типів кількох об’єктів. Найкращий приклад навів сам Андрей Александреску — відеоігри. Під час розрахунку зіткнень об’єктів, якщо стикається астероїд і космічний корабель, результат інший, ніж якщо стикається астероїд і промінь лазера. Очевидно, що всі ці об’єкти мають єдиного спільного нащадка, для якого написана більшість

логіки переміщення чи зникнення з екрана. Але для того, щоб описати цю логіку зіткнення в класі астероїда, необхідно вручну перевірити динамічний тип об'єкта, з яким він стикається. Такий підхід незручний навіть якщо класів небагато, і зовсім не підходить, якщо класів десятки.

Власне, це і є проблема множинної диспетчеризації. Бібліотека Loki23 пропонує узагальнене розв'язання цієї проблеми. По-перше, `StaticDispatcher` реалізує множинну диспетчеризацію для двох аргументів із визначеним під час компіляції набором динамічних типів. Цей тип диспетчера використовується переважно тоді, коли архітектура системи є закритою, тобто всі можливі цільові типи в ієрархії гарантовано відомі розробнику на етапі компіляції програми. Головна перевага полягає в тому, що всі обчислення виконуються під час компіляції. `StaticDispatcher` базується на генерації каскадних серій перевірок типів через безпечний оператор `dynamic_cast`. Використання виразів згортки дозволило замінити громіздкі рекурсивні алгоритми генерації коду на дуже компактний і елегантний для компілятора алгоритм:

```
template <typename BaseLhs, typename BaseRhs, typename
 ↪  ExecutorType, typename... Ts>
struct StaticDispatcherExecutor<DispatchPolicy::FIRST_MATCH,
 ↪  BaseLhs, BaseRhs, ExecutorType, Ts...> {
  template <typename... Functors> static int execute(BaseLhs
   ↪  *lhs, BaseRhs *rhs, Functors... execs) {
    return (... || ([&]() {
              if (auto *l = dynamic_cast<Ts *>(lhs)) {
                return (... || ([&]() {
                          if (auto *r = dynamic_cast<Ts *>(rhs)) {
                            Executor<DispatchPolicy::FIRST_MATCH,
                              ↪  ExecutorType>::execute(l, r,
                              ↪  execs...);
                            return 1;
                          }
                          return 0;
                        }()));
              }
              return false;
            }()));
  }
};
```

Лістинг 16: Генерація статичної диспетчеризації мультиметодів у `Loki23`

Цей шаблонний код генерує та розгортає повну двовимірну матрицю перевірок динамічних типів аргументів і можливих функцій для виклику. У зовнішньому циклі шаблонної згортки ми ітеруємося за всіма можливими підтипами лівого аргументу, а у внутрішній згортці — за можливими підтипами правого аргументу. Якщо функцію знайдено, шаблонний код створює її виклик. Варто звернути увагу, що шаблонний код нічого не викликає, а лише створює код для виклику; це і є шаблонне метапрограмування.

Для більш складних випадків, коли під час компіляції невідомий набір усіх можливих динамічних типів, на жаль, ми не можемо використовувати `StaticDispatcher`. Наприклад, у сучасних додатках система часто є архітектурно відкритою: ієрархія типів може розширюватися плагінами або модулями прямо в процесі виконання програми, або типи можуть бути фізично розбиті по різних динамічних бібліотеках, які нічого не знають одна про одну на етапі компіляції. Тоді статична генерація матриці неможлива. Саме для таких випадків Loki23 реалізує `DynamicDispatcher`. Замість генерації масивного і жорсткого блоку коду на етапі компіляції процес реєстрації функцій-обробників відбувається в процесі виконання програми. Для цього використовується можливість ідентифікації типів часу виконання. Бібліотека використовує клас `std::type_index` для унікальної ідентифікації типів та хеш-таблицю для зберігання зареєстрованих користувачем функцій-обробників:

```
template <DispatchPolicy dispatch_policy, typename
 ↪  NoMatchPolicy, typename... Args>
struct DynamicDispatcher {
  static constexpr std::size_t n_args{sizeof...(Args)};
  using Callback = std::function<void(Args...)>;

  std::conditional_t<dispatch_policy ==
   ↪  DispatchPolicy::FIRST_MATCH,
                 std::unordered_map<std::array<std::type_index,
                    ↪  n_args>, Callback, TypeIndexArrayHash>,

                    ↪  std::unordered_multimap<std::array<std::type_
                    ↪  n_args>, Callback, TypeIndexArrayHash>>
      callbacks_;
};
```

Лістинг 17: Структура `DynamicDispatcher` для реєстрації обробників

мультиметодів під час виконання

Варто також зазначити, що ключем для доступу до цієї хеш-таблиці виступає не один тип, а масив індексів типів `std::array<std::type_index, n_args>`. Це дозволяє використовувати `DynamicDispatcher` для множинної диспетчеризації функцій із довільною кількістю аргументів.

РОЗДІЛ 5. ПРАКТИЧНЕ ЗАСТОСУВАННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ

1. Розробка тестового застосунку з використанням компонентів Loki23

Для оцінювання бібліотеки Loki23 під час реалізації проекту на C++ було створено тестовий застосунок `netprobe`. Це консольна програма для базового аналізу мережевого трафіку, яка може отримувати пакети з мережевого інтерфейсу або з попередньо записаного `pcap`-файлу. Основне призначення застосунку полягає не у створенні повноцінної системи мережевого моніторингу, а у перевірці того, як компоненти Loki23 поводяться в реальній структурі програми, де є динамічні ресурси, поліморфізм, потоки виконання, фабричне створення об’єктів і диспетчеризація за фактичними типами.

Вибір мережевого аналізатора як тестового прикладу є обґрунтованим із кількох причин: по-перше, мережевий трафік природно складається з неоднорідних об’єктів: TCP-пакети, UDP-пакети, DNS-запити, DNS-відповіді, HTTP-запити та HTTP-відповіді; усі вони мають спільні властивості, але потребують різної логіки обробки. По-друге, захоплення трафіку пов’язане з ресурсами, які необхідно відкривати та закривати у правильному порядку. По-третє, отримання пакетів і їх подальший аналіз зручно розділяти на незалежні компоненти. Зрештою, автор цієї дипломної роботи найбільше знайомий саме з мережевими системами.

У загальному вигляді `netprobe` складається з трьох частин:

1. Перша частина відповідає за отримання сирих пакетів. Для цього використовується бібліотека `PcapPlusPlus`, яка надає доступ до мережевих інтерфейсів і файлів захоплення.

2. Друга частина перетворює сирі дані на об’єктну модель застосунку: замість того, щоб працювати з низькорівневими структурами у всіх компонентах програми, `netprobe` створює власні класи пакетів із потрібними для аналізу полями.

3. Третя частина складається зі слухачів, які отримують уже розпізнані пакети і виконують окремі види обробки: збір статистики, виведення в консоль або пошук простих зв’язків між DNS-відповідями та подальшими з’єднаннями.

Центральним елементом моделі даних є абстрактний тип `NetPacket`. Він задає спільний інтерфейс для всіх пакетів, що обробляються застосунком. У ньому зберігаються IP-адреси джерела і призначення, розмір пакета та час отримання. Крім базових полів, клас визначає віртуальні операції для отримання назви типу, створення поліморфної копії та формування текстового подання. Наявність методу поліморфного копіювання є важливою деталлю: пакети передаються через базовий тип, але при цьому слухачі повинні отримувати об’єкт зі збереженим конкретним типом, наприклад DNS або HTTP.

Від `NetPacket` успадковуються класи, що відповідають окремим протоколам. `TcpPacket` містить порти, номери послідовності й підтвердження, розмір вікна та набір прапорців TCP. `UdpPacket` зберігає порти UDP. `DnsPacket` описує DNS-повідомлення: ідентифікатор запиту, домен, тип запису, ознаку відповіді та список отриманих адрес. `HttpPacket` є спеціалізованим TCP-пакетом і додає дані прикладного рівня: HTTP-метод, URL, версію протоколу, заголовок `Host` або код статусу відповіді. Така ієрархія дозволяє обробляти пакети узагальнено, але за потреби звертатися до протокольних деталей.

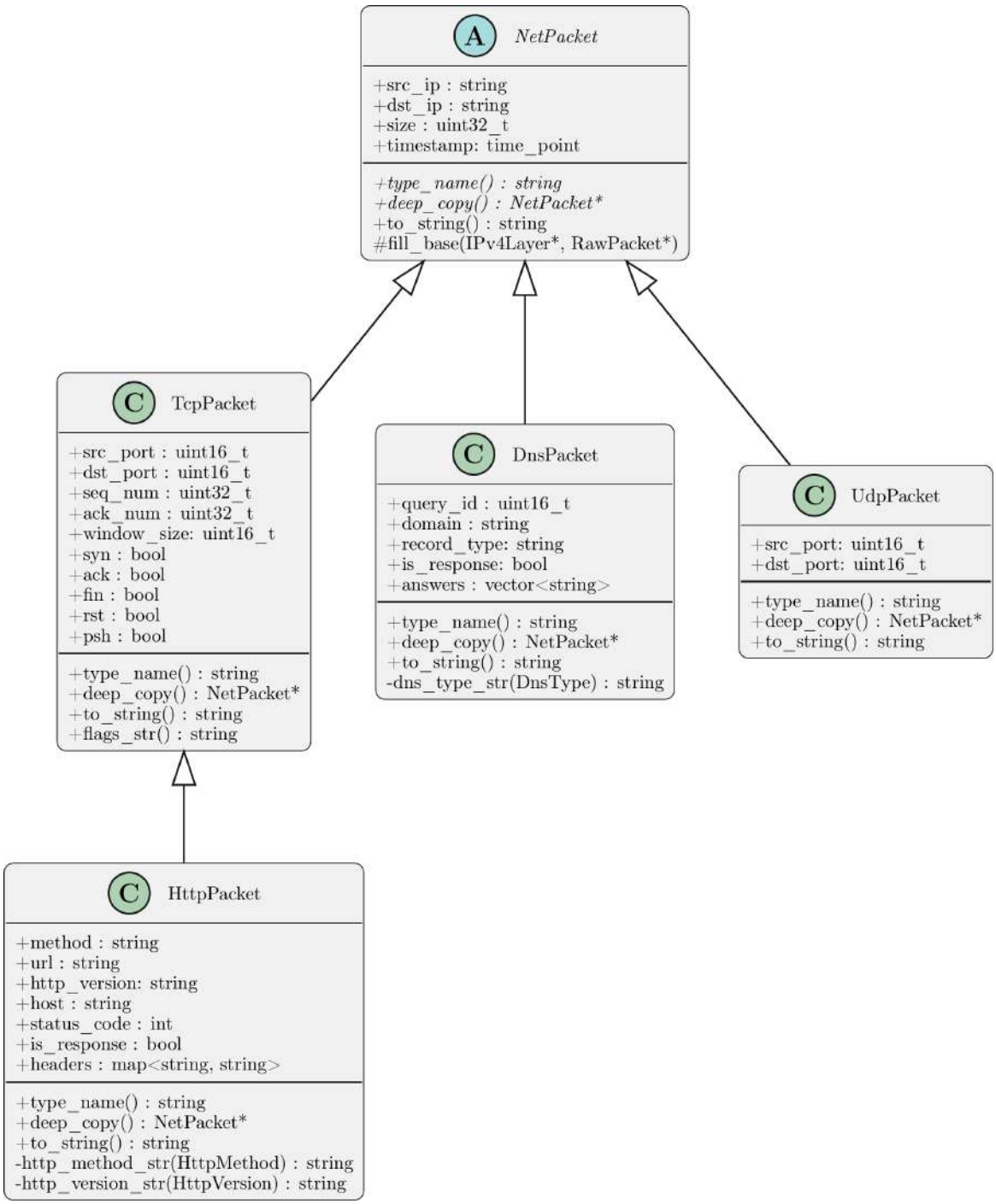

Рис. 4: Логічна структура класів пакетів у `netprobe`

Отримання пакетів винесено в інтерфейс `PacketCapturer`. Він не прив’язаний до конкретного джерела пакетів і визначає лише базові операції відкриття, закриття та передавання пакета підписаним слухачам.

Під час розбору сирого пакета застосунок послідовно намагається створити один із відомих типів: DNS, HTTP, TCP або UDP. Порядок перевірки має практичне значення. HTTP-пакет одночасно є TCP-пакетом, тому його потрібно розпізнавати до загального TCP-випадку. Інакше інформація

прикладного рівня була б втрачена, а пакет потрапив би лише до загальної TCP-статистики. Такий підхід є простим, але достатнім для тестового застосунку, оскільки набір підтримуваних протоколів є обмеженим і явно заданим у коді.

Перший компонент Loki23, який активно використовується в `netprobe`, — це `SmartPtr`. Для різних об'єктів застосунок використовує різні стратегії володіння. Об'єкти захоплення пакетів і слухачі створюються один раз, мають єдиного власника і не повинні копіюватися. Для них використовується стратегія `NoCopy`. Вона робить намір володіння явним і забороняє випадкове копіювання на рівні типів. Це важливо для компонентів, які пов'язані з потоками або зовнішніми ресурсами, оскільки копіювання такого об'єкта могло б спричинити дублювання стану або некоректне завершення роботи.

Але коли ми передаємо пакети користувачам, один розпізнаний пакет може бути потрібний кільком компонентам одночасно. При цьому кожен слухач працює у власному потоці і може завершити обробку в різний момент. Для такого сценарію використовується `SmartPtr<NetPacket, RefCountedMT>`. Потокобезпечний лічильник посилань дозволяє передати одну поліморфну копію пакета кільком отримувачам і звільнити пам'ять лише тоді, коли останній із них завершить роботу з об'єктом.

У цьому прикладі `SmartPtr` демонструє основну ідею проектування на основі стратегій. Поведінка вказівника не є фіксованою для всіх випадків. Вона визначається стратегією, яка передається як параметр шаблону. Завдяки цьому один і той самий інструмент може описувати різні моделі володіння: унікальне некопійоване володіння для компонентів застосунку і спільне потокобезпечне володіння для пакетів, що розсилаються між слухачами.

Другим важливим механізмом Loki23 у застосунку є фабричне створення об'єктів на основі списків типів. У `main.cpp` визначено два набори типів: список слухачів і список джерел пакетів. До першого входять `PacketLogger`, `StatisticsListener` і `SmartListener`, до другого — `LiveCapturer` і `PcapFileCapturer`. Ці набори задано через `type_list`, тобто як структури, відомі на етапі компіляції. Далі вони використовуються для побудови `HierarchyFactory`. Такий підхід зменшує зв'язаність між бізнес-логікою і конкретними класами компонентів.

Використання `type_list` у цьому місці має ще одну перевагу: набір

допустимих реалізацій зафіксовано явно. Якщо потрібно додати новий тип слухача, наприклад аналізатор підозрілих TCP-з'єднань, потрібно реалізувати клас із відповідним інтерфейсом і додати його до списку типів. Подібно можна додати нове джерело пакетів, не змінюючи логіку наявних слухачів. Фабрика і списки типів у `netprobe` працюють як засіб розширення архітектури без дублювання коду створення об'єктів.

Обробка пакетів побудована за моделлю підписників. Клас `PacketCapturer` зберігає список вказівників на `PacketListener` і після розпізнавання пакета розсилає його всім зареєстрованим слухачам. Сам слухач має внутрішню чергу, м'ютекс, умову очікування і робочий потік. Таке рішення зменшує навантаження під час захоплення пакетів. У живому режимі PcapPlusPlus викликає функцію обробки при надходженні пакета. Якщо в цій функції виконувати складний аналіз, захоплення може затримуватися. Тому замість того, щоб виконувати цю логіку в основному потоці, `netprobe` лише створює об'єкт пакета і передає його слухачам, а основна робота виконується окремо. Для тестування Loki23 це корисно, оскільки дозволяє перевірити не тільки базове володіння пам'яттю, а й поведінку об'єктів у багатопотоковому сценарії.

У `netprobe` реалізовано декілька слухачів пакетів, але найцікавіший з них — `SmartListener`. Він реалізує складний аналіз пакетів і використовує мультиметоди Loki23, а саме `DynamicDispatcher`, до якого реєструються обробники для конкретних типів пакетів. Якщо надходить DNS-відповідь, слухач запам'ятовує відповідність між отриманою IP-адресою і доменом. Якщо надходить HTTP-запит або TCP SYN-пакет, слухач намагається зіставити адресу призначення з уже відомим доменом і виводить повідомлення про ймовірне відвідування цього ресурсу. У прикладному коді при цьому не потрібно писати довгий ланцюжок перевірок типу; вибір обробника делеговано диспетчеру.

Крім динамічної диспетчеризації, `SmartListener` використовує `StaticDispatcher` для аналізу пари послідовних пакетів. Слухач зберігає попередній пакет, а при надходженні нового перевіряє комбінації типів. Наприклад, якщо DNS-відповідь безпосередньо передує TCP, UDP або HTTP-з'єднанню до однієї з адрес із відповіді, застосунок повідомляє про такий зв'язок. Цей приклад показує різницю між двома підходами:

`DynamicDispatcher` зручний для реєстрації обробників окремих об'єктів, а `StaticDispatcher` — для наперед відомих комбінацій типів.

Основна функція програми виконує всю координаційну роботу: обробляє ввід користувача, створює необхідні об'єкти і файли, запускає слухачів і отримувачів пакетів.

Отже, застосунок `netprobe` можна розглядати як інтеграційний тест для Loki23. Він одночасно використовує кілька незалежних компонентів бібліотеки і перевіряє їх у спільній роботі. `SmartPtr` керує об'єктами з різними моделями володіння. `type_list` описує набори допустимих реалізацій. `HierarchyFactory` створює компоненти за цими наборами. Диспетчери забезпечують вибір обробника на основі фактичного типу пакета. Якщо будь-який із цих механізмів має помилки в інтерфейсі або життєвому циклі, вони проявляються у досить реалістичному прикладному коді. `netprobe` демонструє практичне використання Loki23 у модульному C++23-застосунку. На його прикладі видно, як шаблонні компоненти можуть допомагати у створенні гнучкої архітектури: ресурси мають явно описане володіння, набір реалізацій задається на етапі компіляції, об'єкти створюються через фабрики, а обробка поліморфних пакетів виконується через диспетчери. У межах дипломної роботи цей застосунок підтверджує, що Loki23 може використовуватися не лише в демонстраційних фрагментах коду, а й у цілісному прикладному сценарії з потоками, зовнішньою бібліотекою і реальною предметною областю.

2. Порівняння продуктивності оригінальної Loki та Loki23

Перед проведенням вимірювань очікувалося, що Loki23 у більшості сценаріїв буде не повільнішою за оригінальну Loki, а в частині тестів покаже кращий результат. Таке очікування пов'язане не лише з адаптацією коду під новіший стандарт C++, а й зі зміною підходу до реалізації окремих механізмів бібліотеки.

Оригінальна Loki створювалася для старіших версій C++, де ще не було багатьох сучасних мовних засобів. Через це значна частина функціональності реалізовувалася через складні шаблонні конструкції, допоміжні типи, рекурсивні шаблонні структури та власні реалізації механізмів, які пізніше з'явилися у стандартній бібліотеці. Такий підхід був ефективним для свого

часу, але він ускладнює код і може створювати додаткові накладні витрати як на етапі компіляції, так і в окремих сценаріях часу виконання.

Loki23 використовує сучасніші можливості C++: варіативні шаблони, стандартні type traits, `constexpr`, move-семантику, чіткіші стратегії володіння ресурсами та модульну організацію коду. Завдяки цьому частину логіки можна виразити простіше, без зайвих проміжних структур і без дублювання того, що вже надає стандартна бібліотека.

Для вимірювань швидкодії були реалізовані декілька тестів, які охоплюють різні аспекти роботи з бібліотекою:

- Копіювання розумного вказівника з підрахунком посилань: тест вимірює створення розумного вказівника, копіювання і розіменування.
- Глибоке копіювання розумного вказівника: перевірка сценарію, у якому під час копіювання створюється окрема копія об’єкта.
- Унікальне володіння ресурсом: вимірювання створення, доступу до об’єкта та знищення розумного вказівника без копіювання.
- Переміщення розумного вказівника: перевірка сценаріїв, у яких Loki23 використовує move-семантику; для оригінальної Loki ці тести позначені як недоступні, оскільки відповідні конструктори переміщення відсутні.
- Робота з ресурсом через політику зберігання файлового дескриптора: тест перевіряє підтримку непокажчикових ресурсів у Loki23; в оригінальній Loki прямого аналога цього механізму немає.
- Багаторазовий доступ до singleton-об’єкта: тест вимірює вартість повторного отримання singleton-екземпляра в щільному циклі.
- Створення об’єкта через фабрику: перевірка пошуку зареєстрованого типу за змінним і сталим рядковим ключем, а також сценарію, де фабрика створюється безпосередньо перед використанням. Це типові сценарії для систем, де конкретний тип визначається під час виконання або під час ініціалізації компонента.
- Перевірка наявності типу у фабриці: додатковий тест API `contains`, який є в Loki23 і відсутній в оригінальній Loki.

- Статична диспетчеризація мультиметодів: оцінка виклику операції для пари типів через механізм статичної диспетчеризації. Основна частина вибору методу в такому підході переноситься на етап компіляції.
- Динамічна диспетчеризація мультиметодів: порівняння функціонально близьких механізмів динамічного вибору методу в оригінальній Loki та Loki23 для змінної послідовності типів і для сталої пари типів. Тест показує вартість вибору операції за динамічними типами об’єктів.

Бенчмарки виконувалися на системі Arch Linux із процесором Intel Core i7-13700H та 32 ГБ оперативної пам’яті; оригінальна Loki компілювалася за допомогою `clang++ 22.1.5` з параметрами `-std=c++17 -O2 -DNDEBUG`, а Loki23 — через CMake/Ninja у конфігурації `Release` з `clang++ 22.1.5`, стандартом C++23 і прапорцями `-O2 -DNDEBUG`. Кожен тест мав один прогрівальний запуск і виконувався у `1000` ітераціях, а час вимірювався за допомогою `std::chrono::high_resolution_clock`. Результати агрегувалися як сумарний час у мілісекундах та експортувалися у CSV, після чого вони оброблялись вручну.

2.1. Результати

Назва тесту	Loki, мс	Loki23, мс	Різниця
Копіювання розумного вказівника з підрахунком посилань	12032.792	13735.749	+14.2%
Глибоке копіювання розумного вказівника	1657.822	1714.282	+3.4%
Унікальне володіння ресурсом	1496.058	1347.902	-9.9%
Багаторазовий доступ до singleton-об’єкта	15007.127	16045.238	+6.9%
Створення об’єкта через фабрику за назвою	24680.464	16367.528	-33.7%

Назва тесту	Loki, мс	Loki23, мс	Різниця
Створення об’єкта через фабрику за сталою назвою	19084.972	15916.295	-16.6%
Ініціалізація фабрики та створення об’єкта	109.444	97.514	-10.9%
Статична диспетчеризація мультиметодів	25533.207	25714.116	+0.7%
Динамічна диспетчеризація мультиметодів	22256.800	9305.764	-58.2%
Динамічна диспетчеризація мультиметодів для сталої пари	23014.188	10550.949	-54.2%

У таблиці наведено зіставні тести, для яких обидві реалізації мають функціонально близький сценарій. За сумарним часом цих тестів оригінальна Loki виконала набір бенчмарків приблизно за 144872.872 мс, тоді як Loki23 — за 110795.337 мс. Отже, за цим набором вимірювань Loki23 є приблизно у 1.31 раза швидшою за оригінальну Loki.

Із десяти зіставних тестів Loki23 показала кращий результат у шести. Оригінальна Loki була швидшою у чотирьох сценаріях, переважно в простих операціях копіювання розумних вказівників, доступі до singleton-об’єкта та статичній диспетчеризації мультиметодів.

2.2. Аналіз результатів

Отримані результати показують, що Loki23 зберігає ефективність оригінальної Loki, але її перевага залежить від конкретного компонента. Найбільший виграш спостерігається не в найпростіших операціях, а в механізмах із більш складною логікою під час виконання: фабриці об’єктів і динамічній диспетчеризації мультиметодів.

У групі тестів розумних вказівників результати неоднорідні. Loki23 швидша у сценарії унікального володіння на 9.9%, але оригінальна Loki швидша в копіюванні вказівника з підрахунком посилань на 14.2% і в глибокому копіюванні на 3.4%. Водночас Loki23 підтримує move-семантику для розумних

вказівників, тому частина практичних сценаріїв володіння ресурсами взагалі не має прямого аналога в оригінальній Loki.

Тест багаторазового доступу до singleton-об’єкта також показав перевагу оригінальної Loki: 15007.127 мс проти 16045.238 мс у Loki23. Різниця становить 6.9%. Це свідчить, що для дуже простого повторного звернення до вже створеного екземпляра стара реалізація має менші накладні витрати.

Найбільш стабільну перевагу Loki23 показала фабрика. У створенні об’єкта за змінною назвою вона швидша на 33.7%, у створенні за сталою назвою — на 16.6%, а в сценарії ініціалізації фабрики разом зі створенням об’єкта — на 10.9%. Це практично важливий результат, оскільки фабрики часто використовуються в системах із динамічним вибором типів: під час обробки конфігурацій, команд, повідомлень або плагінів.

Мультиметоди також показують перевагу Loki23 саме в динамічних сценаріях. У статичній диспетчеризації оригінальна Loki швидша на 0.7%, однак у динамічній диспетчеризації Loki23 швидша на 58.2%, а для сталої пари динамічних типів — на 54.2%. Це найсильніший результат у всьому наборі тестів і головне джерело сумарної переваги Loki23.

Загалом результати показують, що Loki23 не є лише синтаксично оновленою версією Loki. Оригінальна Loki зберігає перевагу в окремих простих мікросценаріях, але Loki23 демонструє кращі результати у фабриці, динамічних мультиметодах і сценаріях, які використовують сучасні можливості C++, зокрема переміщення та узагальнені політики зберігання ресурсів. За сумарним результатом зіставних тестів Loki23 швидша приблизно у 1.31 раза.

3. Аналіз чистоти коду та зручності використання

Окрім продуктивності, важливою характеристикою бібліотеки є її зручність для розробника. Цю метрику можна назвати *Developer Experience*. У випадку Loki та Loki23 цей аспект особливо показовий, оскільки обидві бібліотеки реалізують схожі ідеї узагальненого програмування, але належать до різних етапів розвитку C++. Оригінальна Loki була створена в умовах обмежень старих стандартів C++, тоді як Loki23 спирається на сучасні мовні можливості C++23. Це впливає не лише на швидкодію, а й на читабельність, простоту використання, якість діагностики помилок і підтримуваність коду.

Оригінальна Loki має характерну для свого часу структуру: значна

частина коду організована як набір заголовкових файлів із великою кількістю умовної компіляції та окремими реалізаціями для різних компіляторів. Наприклад, заголовки можуть перенаправляти підключення на `Reference`, `Borland`, `MSVC/1200` або `MSVC/1300` залежно від середовища компіляції. Такий підхід був виправданий історично, оскільки компілятори початку 2000-х мали різний рівень підтримки шаблонів. Проте для сучасного розробника така структура створює додатковий шум: складніше зрозуміти, яка саме реалізація використовується, де знаходиться фактичний код компонента і які частини бібліотеки залишаються актуальними.

Loki23 має чистішу архітектурну організацію. Вона використовує модулі C++23, де головний модуль `loki23` експортує окремі підмодулі. Така структура краще відображає логічні межі бібліотеки. Розробник бачить не набір історичних заголовків, а чітко поділені компоненти з явними точками імпорту. Це покращує навігацію по коду та зменшує кількість випадкових залежностей між частинами системи.

Оригінальна Loki активно використовує шаблонне метапрограмування у стилі старого C++. Наприклад, списки типів реалізуються через рекурсивні структури на основі пари `Head` і `Tail`, а для створення списків типів застосовуються макроси на кшталт `TYPELIST_...`. Для свого часу це було єдиним вдалим рішенням, але сучасному розробнику такий код читати складніше. Логіка операцій над типами часто розподілена між багатьма спеціалізаціями шаблонів, а зміст помилки компіляції може бути прихований у довгому ланцюжку інстанціювань.

У Loki23 списки типів реалізовані через варіативні шаблони, наприклад `type_list<Ts...>`. Така форма безпосередньо виражає ідею списку типів і не потребує макросів для створення списків фіксованої довжини. Операції на кшталт `length`, `type_at`, `index_of`, `contains`, `append` або `erase` залишаються шаблонними, але їхня структура є простішою та ближчою до сучасного C++ стилю. Розробник швидше розуміє, що саме відбувається, оскільки код використовує знайомі механізми стандартної мови.

Подібна різниця помітна і в реалізації `SmartPtr`. У Loki23 стратегії володіння, перевірки, зберігання і конвертації явно розділені, а допустимість певних комбінацій перевіряється через концепти і синтаксис `requires`. Це робить обмеження API видимими безпосередньо в сигнатурі класу.

Наприклад, копіювання доступне лише тоді, коли стратегія володіння підтримує клонування. Такий код краще документує сам себе: частина правил використання не прихована в документації, а виражена в типах і обмеженнях, мовою Modern C++.

З погляду користувача бібліотеки, Loki23 має помітно кращий *Developer Experience* через сучасніший інтерфейс. Замість складної системи заголовних файлів розробник може імпортувати потрібний модуль. У великих проектах така організація також спрощує розуміння того, які саме компоненти потрібні конкретному файлу.

Оригінальна Loki часто вимагає від користувача знання внутрішніх шаблонних домовленостей бібліотеки. Наприклад, для списків типів потрібно знати спеціальні макроси, структуру списків і набір допоміжних метафункцій. Для роботи зі стратегіями `SmartPtr` потрібно розуміти старий стиль дизайну на основі стратегій, у якому помилки використання можуть проявлятися як складні помилки шаблонної інстанціації. Це не робить бібліотеку поганою, але підвищує поріг входу.

У Loki23 той самий підхід до використання стратегій збережено, але він поданий у звичнішій для сучасного C++ формі. Використання концептів, `noexcept`, конструкторів переміщення і стандартних утиліт робить інтерфейс, який надає бібліотека, більш явним. Якщо користувач намагається застосувати неправильну комбінацію стратегій, компілятор має більше інформації для формування точнішого повідомлення про помилку. Це напряму покращує досвід розробника, оскільки він витрачає менше часу на пошук причини проблеми.

Якість повідомлень про помилки є важливою перевагою Loki23. Оригінальна Loki використовує багато вкладених шаблонів, спеціалізацій і допоміжних типів. Коли користувач робить помилку, компілятор може вивести довгий ланцюжок внутрішніх типів, який складно пов’язати з реальною причиною проблеми. Для досвідченого C++ розробника це прийнятно, але для нового користувача бібліотеки такий досвід є важким.

У Loki23 значна частина обмежень описана через концепти. Це дозволяє компілятору раніше відсікати неправильні варіанти використання та показувати помилку ближче до місця, де порушено контракт вимог до класу. Наприклад, якщо стратегія не підтримує потрібну операцію, це може бути виражено як

порушення конкретної вимоги концепту, а не як помилка глибоко всередині реалізації бібліотеки.

Loki23 також краще підходить для розширення. Додавання нової стратегії, нового компонента або нової операції над типами може відбуватися в межах сучасної системи модулів і вимог до типів, прописаних у концептах. Такий підхід зменшує ризик неявних побічних ефектів. Крім того, використання стандартної бібліотеки замість власних історичних аналогів зменшує обсяг коду, який потрібно підтримувати самостійно.

З погляду чистоти коду та досвіду розробника Loki23 є суттєвим кроком уперед порівняно з оригінальною Loki. Основні переваги полягають у модульній структурі, зрозумілішому інтерфейсі, кращій діагностиці помилок і природній підтримці сучасних можливостей C++.

Оригінальна Loki залишається важливою та впливовою бібліотекою, але її код відображає обмеження епохи, у якій вона створювалася. Loki23 зберігає ключові ідеї Loki, зокрема архітектуру на основі стратегій і узагальнене програмування, але подає їх у формі, яка краще відповідає очікуванням сучасного розробника, який звик до використання Modern C++.

ВИСНОВКИ

У роботі було досягнуто наступних результатів:

1. Досліджено еволюцію C++ від класичного C++98 до Modern C++, зокрема вплив варіативних шаблонів, `constexpr`, `consteval`, виразів згортки, концептів, модулів, семантики переміщення та розширень стандартної бібліотеки на проектування узагальнених бібліотечних абстракцій.

2. Розглянуто теоретичні основи проектування на основі стратегій Александреску та показано, що цей підхід дає змогу конфігурувати поведінку класів на етапі компіляції, розділяти незалежні аспекти поведінки й уникати накладних витрат класичного об’єктно-орієнтованого поліморфізму.

3. Проаналізовано історичний вплив бібліотеки Loki на розвиток C++ і показано, що її ідеї у сфері списків типів, перевірок часу компіляції, розумних вказівників, фабрик об’єктів, функціональних об’єктів та узагальненого програмування стали важливим етапом переходу від C++98 до сучасного C++.

4. Сформульовано та реалізовано задачу розробки бібліотеки `Loki23`, яка переосмислює основні компоненти Loki засобами C++23: списки типів на основі варіативних шаблонів, генерацію ієрархій, фабричне створення об’єктів, розумний вказівник зі стратегіями володіння, перевірки, зберігання та конвертації, модернізований `Singleton`, а також статичну і динамічну диспетчеризацію мультиметодів.

5. Проектування вилилось у модульну архітектуру бібліотеки `Loki23`, пристосовану до розширення та супроводу: історичні макроси, рекурсивні шаблонні конструкції та неявні вимоги до типів замінено варіативними шаблонами, концептами, `requires`-обмеженнями й стандартними засобами мови.

6. Розроблено тестовий застосунок `netprobe`, що демонструє практичне використання компонентів `Loki23` у цілісному прикладному сценарії

аналізу мережевого трафіку з файлу або мережевого інтерфейсу, зокрема в умовах роботи з поліморфною моделлю даних, потоками виконання, зовнішньою бібліотекою та реальними ресурсами.

7. Здійснено порівняння оригінальної Loki та `Loki23` за продуктивністю, архітектурною організацією та зручністю використання. Із десяти зіставних тестів `Loki23` показала кращий результат у шести, а за сумарним часом виконання набору бенчмарків була приблизно у 1.31 раза швидшою за оригінальну Loki. Також встановлено, що явні контракти на основі концептів, модульна структура та використання стандартних механізмів C++23 покращують типобезпеку, діагностику помилок, підтримуваність і досвід розробника.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Peter Dimov, Howard E. Hinnant та Dave Abrahams. *The Forwarding Problem: Arguments*. Тех. звіт. N1385=02-0043. ISO/IEC JTC1/SC22/WG21 - C++ Standards Committee, вер. 2002. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1385.htm.

[2] Gabriel Dos Reis, Bjarne Stroustrup та Jens Maurer. *Generalized Constant Expressions — Revision 4*. Тех. звіт. N2116-06-0186. ISO/IEC JTC1/SC22/WG21, листоп. 2006. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2116.pdf.

[3] Erich Gamma та ін. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2.

[4] Douglas Gregor. *A Proposal to add a Polymorphic Function Object Wrapper to the Standard Library*. Тех. звіт. N1402=02-0060. ISO/IEC JTC1/SC22/WG21 - C++ Standards Committee, жовт. 2002. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1402.html.

[5] Douglas Gregor, Jaakko Järvi та Gary Powell. *Variadic Templates (Revision 3)*. Working Paper N2080. ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee, вер. 2006. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf.

[6] Howard E. Hinnant, Peter Dimov та Dave Abrahams. *A Proposal to Add Move Semantics Support to the C++ Language*. Document N1377=02-0035. ISO/IEC JTC1/SC22/WG21, вер. 2002. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm.

[7] Brian W. Kernighan та Dennis M. Ritchie. *The C Programming Language*. Second. Englewood Cliffs, NJ: Prentice Hall, 1988. ISBN: 978-0131103627.

[8] Zezheng Li. *42% Boost in Compilation Efficiency! A Practical Analysis of C++ Modules*. Alibaba Cloud Community Blog. Describes production deployment of C++20 Modules in Alibaba Hologres, reporting a 42% reduction in compilation time. Alibaba Cloud Intelligence Group, лют. 2025. URL: https://www.alibabacloud.com/blog/42%25-boost-

in-compilation-efficiency-a-practical-analysis-of-c%2B%2B-modules_601974 (дата зверн. 21.03.2025).

[9] Baishakhi Ray та ін. «A large-scale study of programming languages and code quality in GitHub». В: *Commun. ACM* 60.10 (вер. 2017), с. 91—100. ISSN: 0001-0782. DOI: 10.1145/3126905. URL: https://doi.org/10.1145/3126905.

[10] Douglas C. Schmidt та ін. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Т. 2. Pattern-Oriented Software Architecture. John Wiley & Sons, 2000. ISBN: 978-0471606956.