Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

# Курсова робота

першого (бакалаврського) рівня вищої освіти

на тему: «**СТРАТЕГІЇ БАГАТОПОТОКОВИХ ІНТЕЛЕКТУАЛЬНИХ УКАЗНИКІВ**»

Виконав: студент 3-го року навчання,
Освітньої програми «Інженерія програмного забезпечення», 121
Житнецький Олексій Олександрович

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

Київ – 2025

# Зміст

# 1. Вступ

У сучасному програмуванні дедалі більшої ваги набувають питання продуктивності, безпеки та масштабованості програмного забезпечення. З огляду на стрімкий розвиток апаратних технологій, одним із ключових чинників, що впливають на ефективність виконання програм, є здатність максимально використовувати ресурси сучасних багатоядерних процесорів. Паралельно з цим зростає і складність проектування архітектури програм, де важливими є як коректне управління потоками, так і забезпечення безпечного доступу до спільних ресурсів.

Актуальність теми полягає у необхідності дослідження та впровадження ефективних методів до побудови потокобезпечного програмного забезпечення у C++, адже попри наявність розвинених інструментів у сучасному стандарті мови, практичне застосування таких механізмів, як інтелектуальні указники, примітиви синхронізації та стратегії поведінки, досі викликає низку труднощів у розробників. Саме тому дослідження цих засобів та їх узгоджене використання в межах єдиного програмного рішення є важливим як із теоретичного, так і з практичного погляду.

Метою курсової роботи є знаходження ефективних підходів до проектування потокобезпечного програмного забезпечення в C++ з урахуванням принципів сучасного управління пам’яттю та метапрограмування шаблонів.

Для досягнення поставленої мети були визначені наступні завдання:

1. Визначити особливості реалізації багатопоточності у C++ та засобів управління потоками.
2. Проаналізувати фундаментальні механізми синхронізації та їх вплив на продуктивність.

3. Систематизувати проблеми керування пам’яттю у багатопотоковому середовищі та розглянути можливості їх вирішення.
4. Розглянути інтелектуальні указники STL та їхню роль у безпечному управлінні ресурсами.
5. Дослідити взірець «стратегія» як інструмент для побудови гнучкої поведінки об’єктів.
6. Реалізувати демонстраційний застосунок із використанням розглянутих концепцій.

Об’єкт дослідження – процес проектування потокобезпечного програмного забезпечення мовою C++ з використанням засобів STL.

Практичне значення одержаних результатів полягає в розробці демонстраційного архіватора, що використовує сучасні засоби багатопоточності та керування пам’яттю і може бути основою для створення складніших прикладних програм у подальших дослідженнях та розробці.

# 2. Основи багатопоточності у C++

## 2.1. Потік керування (control flow)

У однопотокових імперативних програмах потік керування визначає послідовність виконання інструкцій у коді. Програма починається з точки входу (традиційно функція main), після чого виконується у чіткому порядку, інструкція за інструкцією, зверху донизу. Водночас, існує ряд операторів потоку керування – умовні розгалуження (if, else та switch-case), цикли (for, while, do-while) та переходи (continue, break, goto та виклики функцій, що є неявними goto із запам'ятовуванням адреси повернення для подальшого її використання у return, що теж є неявним goto).

Перш ніж перейти до порівняння потоку керування в однопотокових і багатопотокових програмах, варто уточнити, що саме розуміється під терміном "багатопоточність" у контексті цієї роботи. Як зазначає Ентоні Вільямс, слід чітко розрізняти два підходи до паралельного виконання: справжню (апаратну) багатопоточність і симульовану [1].

У системах із апаратною багатопоточністю, що передбачає наявність декількох процесорів або багатоядерних процесорів, декілька потоків можуть фізично виконуватися одночасно – кожен на окремому ядрі. Це і є справжнє паралельне виконання, у цьому випадку кожне ядро процесора обробляє свою частину роботи, а обчислення дійсно відбуваються одночасно паралельно, без чергування.

Натомість у системах із симульованою багатопоточністю, які мають лише одне обчислювальне ядро, паралельність досягається шляхом частого перемикання контексту – операційна система просто чергує виконання задач, зберігаючи й відновлюючи стан виконання кожної з них. Це створює лише ілюзію одночасного виконання, тоді як насправді у будь-який момент часу виконується лише один потік. Контекстне перемикання також має свою

вартість: щоразу потрібно зберігати стан поточної задачі, завантажувати нову, можливо очищувати кеш – усе це уповільнює виконання.

Отже, у межах цієї роботи уся увага буде зосереджена виключно на апаратній багатопоточності, адже саме така форма багатопоточності є найбільш релевантною у розробці високопродуктивного програмного забезпечення.

Відповідно, повертаючись назад концепту потоку керування, у багатопотокових програмах, за умови що ми дивимось на кожен потік в ізоляції, правила потоку керування залишаються незмінними і інструкції виконуються так само. Однак уся користь, але й складність багатопоточності полягає у можливості комп'ютера виконувати декілька інструкцій одночасно. Відповідно, хоча кожен окремий потік дотримується явно прописаних правил виконання інструкцій та зберігає стандартний потік керування, багатопоточна програма як композит порушує цей потік керування: у ній неможливо достеменно визначити єдиний порядок виконання інструкцій, адже цей порядок тепер залежить від завантаженості середовища виконання, реалізації операційної системи та навіть від типу апаратного забезпечення, на якому ця програма виконується.

Отже, у силу перелічених вище причин, у розробці багатопотокових застосунків програміст не може покладатися виключно на розуміння правил потоку виконання, і має натомість узяти на себе безпосередню відповідальність за забезпечення цілісності даних та стабільності виконання програми.

## 2.2. std::thread

std::thread – основний та найфундаментальніший клас для створення та керування потоками, доданий до мови у стандарті C++11 [2]. Запустити новий потік дуже просто – достатньо просто створити екземпляр класу std::thread, передавши йому в конструктор об'єкт, що можна викликати (callable object),

що і буде корисним навантаженням потоку (Лістинг 1). Таким об’єктом може бути як звичайна функція, так і лямбда-вираз, метод класу (але тоді додатковим параметром у конструкторі потоку необхідно передати адресу екземпляру класу, на якому метод викликається, де-факто “this”) чи навіть об’єкт-функтор, тобто екземпляр класу із довизначеним (overloaded) оператором operator().

*Лістинг 1*

```
#include <iostream>
#include <thread>

static void greeting(const char *name = nullptr)
{
    std::cout << "Hello, " << ((name == nullptr) ? "World" : name)
              << '!' << std::endl;
}

int main()
{
    std::thread thread(greeting, "Oleksiy");
    greeting();

    thread.join();
    return 0;
}
```

Розглянемо детальніше, як працює клас std::thread, починаючи з передачі аргументів. Першим аргументом, що передається у конструктор, як уже було сказано вище, є власне процедура (функція, callable object), яку ми хочемо запустити на виконання у новому потоці. Усі наступні аргументи – це аргументи обраної процедури. Дуже важливо зазначити, що аргументи, передані у конструкторі std::thread, потрапляють до процедури шляхом копіювання їх значення – відповідно, отримуємо абсолютно коректну та очікувану поведінку для примітивних (plain old data) типів (копіюється власне значення) та указників (копіюється адреса –значення указника). Водночас, виникають проблеми із відсилками, адже навіть якщо сигнатура процедури очікує відсилку (не важливо, сталу, чи ні), при передачі відсилки у конструктор std::thread, її значення (тобто значення власне об’єкту, до якого вона прив’язана) буде просто скопійовано. Отже, для правильної передачі відсилок у конструкторі std::thread, їх слід обгортати у std::ref чи std::cref відповідно до їх сталості [3].

Ще одною потенційно неочевидною частиною класу std::thread є власне процес створення та запуску нового потоку. У C++, як і у інших мовах, кожен потік має свій окремий стек викликів (call stack), розмір якого за замовчуванням залежить від компілятора, але зазвичай дорівнює 1-2 мегабайти і є, на замовлення програміста, змінним під час компіляції програми. Отже, виділення такого блоку пам'яті для створення нового потоку несе за собою накладні витрати на час виконання та пам'ять. Більш того, навіть після виділення стеку, виконання нового потоку не гарантовано почнеться одразу – власне запуск потоку тепер делегується операційній системі. Навіть тривіальний приклад, наведений вище у Лістингу 1, демонструє порушення у багатопотокових програмах стандартного потоку виконання: "Hello, World!" майже завжди буде виведено перед "Hello, Oleksiy!".

Останньою критично важливою складовою роботи із класом std::thread є розуміння його методів joinable, join та detach. Справа у тому, що усі екземпляри std::thread створені у коді обов'язково мають бути або приєднані назад до батьківського потоку (join), або від'єднані від нього (detach) до завершення його виконання. Відповідно, метод joinable перевіряє, чи можна приєднати потік назад до батьківського. Цей метод повертає істину лише у випадку, коли екземпляр на якому він викликаний ще не був ані приєднаний, ані від'єднаний. Важливо зазначити, що спроба приєднати або від'єднати потік, для якого joinable повертає хибу завжди призводить до аварійної ситуації (exception).

У свою чергу метод join призупиняє виконання батьківського потоку та чекає на завершення потоку-екземпляру на якому цей метод викликаний. Мета методу join очевидна – забезпечити легкий спосіб гарантувати завершення створеного потоку до певного моменту у коді батьківського потоку, таким чином захищаючи увесь наступний код від неоднозначностей багатопоточності та відновлюючи стандартний потік керування програми.

Зрештою, метод detach повністю від'єднує екземпляр від ієрархії потоків – такий потік неможливо ані приєднати, ані дочекатись на завершення його виконання. Втім, метод detach використовується саме з цією метою – створити фоновий потік (daemon thread), що не примусить основний потік під час свого завершення чекати на завершення фонового потоку. Такі потоки часто виконують роль моніторингу системи та працюють нескінченно протягом усього часу роботи основної програми, а отже не мають ані результату, ані причини затримувати завершення основного потоку.

## 2.3. std::future

Окрім власне прямого контролю над потоками через std::thread, стандарт C++11 також надає більш високорівневий спосіб роботи з асинхронністю, а саме із результатами асинхронних операцій. Цим способом якраз і є std::future – клас-шаблон, що надає механізм для отримання значення, обчисленого асинхронно (в іншому потоці виконання або в інший момент часу) [4]. Важливо зазначити, що сам по собі std::future не відповідає за запуск нових потоків та не має жодної інформації про процедуру, що виконується асинхронно, окрім її результату. Відповідно, для того, щоб оптимально працювати зі std::future, стандарт надає три основних абстракції – std::promise, std::packaged_task та std::async.

Почнемо із найбільш низькорівневого інструменту – std::promise. Це шаблон-класу (std::promise<T>, де T – тип загорнутого результату), який дозволяє явно встановлювати значення результату виконання потоку через метод set_value [5]. Для зручності обробки помилок, стандартною бібліотекою також надається метод set_exception, за допомогою якого програміст може передати інформацію про аварійну ситуацію на потоці назад до батьківського потоку. Розглянемо приклад (Лістинг 2), що демонструє використання std::promise. Як бачимо, корисним навантаженням потоку thread є лямбда-вираз, що захоплює promise як lvalue reference для подальшого присвоєння

йому результату за допомогою методу set_value. Надалі, завдяки зв'язку між promise та result, батьківський потік може дочекатися на результат створеного ним потоку через виклик методу get на result.

*Лістинг 2*

```
#include <string>
#include <thread>
#include <future>
#include <iostream>

static std::string get_greeting(const char *name = nullptr)
{
    return std::string("Hello, ") + ((name == nullptr ? "World" : name)) + '!';
}

int main()
{
    std::promise<std::string> promise;
    std::future<std::string> result = promise.get_future();
    std::thread thread(
        [&promise](const char *name = nullptr) {
            promise.set_value(get_greeting(name));
        },
        "Oleksiy"
    );

    std::cout << "Greeting is \"" << result.get() << '\"' << std::endl;
    thread.join();
    return 0;
}
```

Варто звернути увагу, що, оскільки std::promise є низькорівневим інструментом std::future, екземпляри цього класу не зберігають жодної інформації про власне процедуру, що запускається на новому потоці, натомість зберігаючи виключно результат виконання процедури та, за потреби, інформацію про аварійну ситуацію. Така відсутність інформації про виконувану процедуру має як позитивні, так і негативні наслідки:

- Незалежність виконуваної процедури та екземпляру std::promise є беззаперечною перевагою у випадках, коли програміст має на меті реалізувати універсальний інструмент, абстрагований від єдиної, конкретної процедури. При роботі із std::promise, процедури можуть бути довільними, за умови, що тип їх результату є однаковим та збігається з або неявно зводиться до типу T, яким інстанціонований екземпляр std::promise.

- Водночас, така незалежність є недоліком там, де вона надлишкова та небажана для програміста. Це призводить до ускладнення розуміння програми, оскільки часто може бути неочевидно, результат якої саме процедури передається через певний екземпляр std::promise.

Наступним, більш високорівневим методом роботи зі std::future є шаблон класу std::packaged_task [6]. Найбільшою відмінністю std::packaged_task від попередньо розглянутого std::promise є чітке, явне зв'язування екземпляру цього класу із процедурою, яку ми хочемо асинхронно виконати.

Загалом, за способом ініціалізації, std::packaged_task багато чим нагадує std::function – екземпляри обох шаблонів є функторами, а тип T для них є загальним типом усієї процедури, тобто окрім типу результату містить й інформацію про кількість аргументів процедури та тип кожного з них [7]. Водночас очевидна й різниця:

- Для std::packaged_task тип результату operator() завжди void, оскільки справжній результат не повертається одразу, а записується у вбудований std::future.
- На відміну від std::function, std::packaged_task має видалений копіювальний конструктор. Відповідно, при запуску нового потоку, екземпляр std::packaged_task потрібно передавати у конструктор std::thread використовуючи семантику переміщень [8].

Розглянемо принцип застосування std::packaged_task на прикладі уже загаданої вище у Лістингу 2 програми get_greeting (Лістинг 3). Бачимо, що процедура get_greeting, результат якої ми хочемо обчислити на потоці thread та повернути із нього, просто "обгортається" у екземпляр std::packaged_task, який потім пересувається у новий потік при його запуску.

*Лістинг 3*

```
#include <string>
#include <thread>
```

```cpp
#include <future>
#include <utility>
#include <iostream>

static std::string get_greeting(const char *name = nullptr)
{
    return std::string("Hello, ") + ((name == nullptr ? "World" : name)) + '!';
}

int main()
{
    std::packaged_task<std::string(const char*)> task(get_greeting);
    std::future<std::string> result = task.get_future();
    std::thread thread(std::move(task), "Oleksiy");

    std::cout << "Greeting is \"" << result.get() << '\"' << std::endl;
    thread.join();
    return 0;
}
```

Отже, ключовою перевагою std::packaged_task є суттєво простіша послідовність запуску процедури на асинхронне виконання та зменшення кількості зовнішніх змінних (не потрібно створювати окремий об'єкт std::promise, прив'язувати його до std::future та передавати (зазвичай у лямбда-виразі, через захоплення контексту) у конструктор std::thread). Тим не менш, навіть при використанні std::packaged_task, розробник мусить явно створити екземпляр класу std::thread і не забути приєднати його до головного потоку після отримання результату з std::future.

У випадку, коли програміст хоче мінімізувати кількість допоміжних об'єктів на допомогу приходить std::async – інструмент, що надає найвищий рівень абстракції від низькорівневих технологій [9]. На відміну від двох інших попередньо розглянутих засобів, std::async є шаблоном функції, а не класу. Механізм використання std::async простий – розробнику усього лише необхідно надати стратегію запуску процедури та власне процедуру із усіма необхідними для її виклику параметрами.

Стандартна бібліотека C++ надає дві стратегії запуску: std::launch::async – гарантований запуск процедури у новому потоці та std::launch::deferred – відкладений запуск процедури на батьківському потоці, що відбудеться лише після виклику методів get чи wait. Варто зазначити, що стратегії запуску можуть бути поєднані за допомогою оператора побітової диз'юнкції, однак у

цьому випадку остаточний вибір стратегії запуску залежить від компілятора. Важливо також зазначити, що, оскільки std::async відповідає безпосередньо за запуск потоку (за умови використання відповідної стратегії запуску), то й приєднання цього потоку назад до батьківського також є автоматичним.

Продемонструємо переваги std::async на усе тому ж прикладі – програмі get_greeting (Лістинг 4). Бачимо ще простіший та більш прямолінійний синтаксис – шаблон функції std::async перебирає на себе відповідальність за занесення результату в std::future та створення й приєднання потоку нового потоку, на якому процедура get_greeting запускається.

*Лістинг 4*

```
#include <string>
#include <thread>
#include <future>
#include <iostream>

static std::string get_greeting(const char *name = nullptr)
{
    return std::string("Hello, ") + ((name == nullptr ? "World" : name)) + '!';
}

int main()
{
    std::future<std::string> result = std::async(
        std::launch::async,
        get_greeting,
        "Oleksiy"
    );
    std::cout << "Greeting is \"" << result.get() << '\"' << std::endl;
    return 0;
}
```

## 2.4. Фундаментальні засоби синхронізації

У багатопотокових програмах синхронізація необхідна для координування доступу до спільних ресурсів. Без належної синхронізації одночасний доступ декількох потоків до змінної або структури даних призводить до “стану гонки” (race condition), непередбачуваних помилок та порушення цілісності даних.

Основна мета синхронізації полягає в тому, щоб гарантувати коректний порядок виконання операцій над спільними ресурсами, забезпечуючи

взаємовиключення (mutual exclusion) або атомарність (atomicity) доступу. У C++11 та новіших стандартах запропоновано декілька фундаментальних інструментів для досягнення цієї мети: std::mutex та std::atomic.

std::mutex – це базовий примітив синхронізації, який надає механізм взаємовиключення: лише один потік за раз може захопити м'ютекс, тоді як інші змушені чекати на його звільнення [10]. Клас std::mutex визначає методи:

- lock – потік, що викликає метод захоплює м'ютекс, якщо той вільний, інакше блокується до звільнення м'ютексу.
- unlock – потік, що викликає метод звільняє м'ютекс, дозволяючи іншим потокам його захопити.
- try_lock – потік, що викликає метод намагається захопити м'ютекс, однак у випадку, якщо він недоступний, метод явно інформує про це потік шляхом повернення false.

Втім, ручне керування захопленням та звільненням м'ютексів є дуже схожим на застосування new та delete при роботі з пам'яттю у сенсі, що воно є вразливим до помилок, спричинених неуважністю програміста – розробник може банально забути викликати unlock. Одним з найкращих способів подолання такої проблеми є застосування механізму RAII (resource acquisition is initialisation), що гарантує автоматичне звільнення ресурсів при виклику деструктора. У стандартній бібліотеці C++11, інструментом, що реалізує цей взірець є std::lock_guard [11]. Спосіб роботи цього інструмента абсолютно інтуїтивний – для створення об'єкту std::lock_guard достатньо передати м'ютекс, який ми захоплюватимемо. Далі, std::lock_guard самостійно (відповідно до взірця RAII) заблокує потік до захоплення м'ютексу, після чого виконається код, який програміст хоче синхронізувати. Далі, після завершення виконання функції, екземпляр класу std::lock_guard буде видалено, під час чого й викличеться деструктор класу, що звільнить м'ютекс.

Попри беззаперечне забезпечення потокобезпочності програми, блокування потоків за допомогою механізму взаємовиключення є доволі дорогим рішенням з точки зору продуктивності програми і насправді має застосовуватись виключно у випадках, де йому немає альтернатив. Однак, для великої кількості примітивних операцій така альтернатива існує – це std::atomic – шаблон класу, який гарантує, що читання та запис його внутрішньої змінної відбуватимуться безпечно, без переривання іншими потоками [12]. Використання std::atomic є суттєво ефективнішим, аніж використання м'ютексів, адже атомарні операції зазвичай реалізуються апаратними засобами процесора.

Тим не менш, важливо пам'ятати, що std::atomic призначений для простих операцій на простих структурах даних. При спробі його застосування на складніших типах (зазвичай більших за 8 байтів), реалізація std::atomic змінюється та починає використовувати внутрішні м'ютекси, суттєво знижуючи продуктивність. Для перевірки того, що конкретний std::atomic працюватиме без блокування, можна використати метод is_lock_free чи звернутись до статичного поля is_always_lock_free.

# 3. Засоби керування пам’яттю

## 3.1. Структура пам’яті – стек та купа

Усі програми, що виконуються в сучасних операційних системах, послуговуються оперативною пам’яттю для зберігання своїх даних. Відповідно, у всіх сучасних мовах програмування, зокрема у C++, існує чіткий поділ пам’яті на два основних сегменти – стек (stack) і купу (heap), що принципово відрізняються за способом виділення, управління та використання пам’яті.

Стек – це ділянка пам’яті, що використовується для зберігання локальних змінних, аргументів та адрес повернення функцій програми, та керується механізмом "останнім прийшов – першим пішов" (LIFO – Last In First Out). Із особливостей стеку доречно перерахувати наступні:

- Пам’ять на стеку виділяється та звільняється автоматично (неявно) під час виконання програми.
- Висока швидкість доступу до об’єктів завдяки безпосередньому управлінню через указник стеку (stack pointer).
- Обмежений розмір (зазвичай 1-2 мегабайти, залежно від операційної системи та налаштувань компілятора).
- Як вже було зазначено у попередньому розділі, кожен потік у багатопоточних програмах має свій власний стек.

На противагу стеку існує купа – область пам'яті, яка виділяється динамічно та час існування об’єктів якої напряму контролюється програмістом. Виділення пам’яті на купі здійснюється за допомогою оператора new, а звільнення – за допомогою delete. У порівнянні з операціями над стеком, процес виділення чи звільнення пам’яті на купі є суттєво складнішим та повільнішим, оскільки він керується напряму операційною

системою, що зберігає таблицю усіх блоків виділеної пам’яті на пристрої. Однак, попри ці недоліки, купа часто є незамінною у силу ряду причин:

- Гнучка в управлінні – програміст самостійно керує терміном життя (lifetime) об’єктів.
- Може зберігати набагато більші об’єми даних та власне й призначена для зберігання таких даних.
- Дозволяє працювати зі структурами даних, які неможливо елегантно реалізувати на стеку – зв’язні списки, дерева тощо.

## 3.2. Указники

Указник (pointer) – це вид змінної, що зберігає як своє значення адресу іншого об'єкта в пам’яті програми. Таким чином, указник є посередником, що забезпечує непрямий доступ до значення об’єкту, на який він вказує. Указники в C++ відіграють важливу роль, адже вони дозволяють працювати з динамічно виділеною пам’яттю (результатом успішного виділення пам’яті на купі завжди є указник на початок виділеного блоку), створювати динамічні структури даних та передавати дані у функції з можливістю зміни цих даних зсередини функції.

Розглянемо класичну задачу обміну значень двох змінних типу int. Стандартна функція swap повинна взяти дві змінні й поміняти місцями їх значення. Якщо спробувати реалізувати swap без указників (Лістинг 5), значення будуть передаватись у функцію за значенням, що означатиме копіювання цих значень у локальні змінні функції. Відповідно, зміни локальних копій не вплинуть на оригінальні змінні.

*Лістинг 5*

```
void swap(int lhs, int rhs)
{
    int tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Тут на допомогу і приходять указники (Лістинг 6). Вони дозволяють передавати у функцію адреси оригінальних змінних, що дає можливість змінювати самі ці змінні безпосередньо в пам’яті, де вони були створені. Функція приймає два указники, розіменовує їх, і виконує обмін значень за цими адресами. Таким чином, після виконання функції swap, оригінальні змінні мають оновлені значення.

*Лістинг 6*

```
void swap(int *lhs, int *rhs)
{
    int tmp = *lhs;
    *lhs = *rhs;
    *rhs = tmp;
}
```

## 3.3. Відсилки

Відсилка (reference) – це альтернативне ім'я (alias) вже існуючої змінної. Вона дозволяє створювати додаткове ім’я, через яке можна напряму отримати доступ до значення цієї змінної. Відсилки завжди повинні бути ініціалізовані при оголошенні і після цього не можуть бути змінені, тобто не можуть бути переприв’язані до іншого об'єкту.

Насправді, з точки зору їх низькорівневої імплементації, відсилки – це синтаксичний цукор – спеціальний вид сталих указників, що усього лише створюють ілюзію прямого доступу до значення об’єкту із яким вони зв’язані.

Повернемося до функції swap та реалізуємо її через відсилки (Лістинг 7). Як бачимо, окрім трохи зміненої сигнатури, де тепер параметри передаються відсилками, код реалізації ідентичний коду у Лістингу 3. Водночас, на відміну від прикладу у Лістингу 3, реалізація через відсилки коректно замінює значення зовнішніх змінних.

*Лістинг 7*

```
void swap(int &lhs, int &rhs)
{
    int tmp = lhs;
    lhs = rhs;
    rhs = tmp;
```

```
}
```

## 3.4. Проблеми керування пам’яттю у багатопотоковому середовищі

У багатопотокових програмах правильне керування пам’яттю стає значно складнішим та критично важливим завданням. Використання “сирих” (raw) указників та відсилок має низку підводних каменів навіть у однопотокових застосунках. Так, метр стандартної бібліотеки C++ Меєрс наводить у своїй книзі аж 6 серйозних недоліків використання сирих указників [13]. Відповідно, логічно припустити, що усі ці проблеми стають ще більш відчутними та болючими у багатопотоковому контексті.

Напевно найвідомішою проблемою, що може трапитись під час роботи із сирими указниками є витік пам’яті (memory leak). Причинами для витоку пам’яті часто є неправильне або взагалі відсутнє видалення динамічно виділених ресурсів, а також неочевидне створення нових ресурсів у певній функції (виділення нової пам’яті) та неявне перенесення відповідальності за звільнення пам’яті на користувача цієї функції. У багатопотоковому середовищі, коли декілька потоків працюють із одним блоком динамічно виділеної пам’яті, уникнення цієї проблеми стає ще складнішим, як мінімум тому що програміст не може просто звільнити пам’ять після виконання роботи у кожному потоці, а має натомість знищити динамічний об’єкт лише тоді, коли на нього більше немає посилань. Це – вкрай необхідна функціональність, якої сирі указники очевидно не надають.

# 4. Інтелектуальні указники. Їхня реалізація у STL

## 4.1. Ключові характеристики інтелектуальних указників

Інтелектуальні указники (smart pointers) можна охарактеризувати як спеціальні класи, об’єкти яких імітують поведінку звичайних указників, але надають додаткові можливості та поведінку для безпечного та ефективного управління динамічними ресурсами. Їх поява стала природною відповіддю на низку проблем, властивих небезпечно керованим вручну указникам, таких як витоки пам’яті, подвійне вивільнення ресурсів, “завислі указники” (dangling pointers) тощо. Також, ідеологічно, інтелектуальні указники майже завжди реалізовують уже згаданий у розділі 2 взірець RAII.

Головна ідея та мета існування інтелектуального указника полягає в автоматизації управління ресурсами: замість того, щоб явно викликати delete для звільнення пам’яті, розумний указник самостійно звільняє ресурс у відповідний момент часу, наприклад при виході за межі області видимості. Такий підхід значно знижує ризики програмних помилок, підвищує надійність коду і спрощує його подальший супровід.

Іншою суттєвою перевагою є захист від подвійного вивільнення. При використанні сирих указників навіть у досвідчених розробників нерідко виникають ситуації, коли один і той самий ресурс ненароком (часто неявно, при автоматичних викликах різних деструкторів) звільняється декілька разів, що призводить до аварійного завершення або, ще гірше, непередбачуваної поведінки програми. Натомість, інтелектуальні указники послуговуються моделями володіння, що запобігають таким помилкам: кожен ресурс має чітко визначеного власника або множину координованих власників.

Наостанок, третьою беззаперечною перевагою є уникнення завислих указників – проблеми, що виникає, коли область пам’яті, на яку указник посилається, вже вивільнена, але сам він продовжує існувати і потенційно використовується в подальшому. Відповідно, на противагу сирим указникам,

інтелектуальні або автоматично обнуляються після знищення ресурсу, або мають механізми безпечної перевірки актуальності посилання.

Однак, перед тим, як перейти до розгляду конкретних реалізацій, що надаються C++ STL, критично важливо розібрати детальніше, що таке семантики (або ж моделі) володіння та які конкретні їх реалізації використовуються інтелектуальними указниками.

Отже, загалом, володіння об’єктом буквально означає відповідальність за життєвий цикл об’єкта. У контексті динамічного управління пам’яттю володіння означає, що певний об’єкт (або множина об’єктів у певних випадках) має повноваження на створення, використання і знищення іншого об’єкта, яким цей об’єкт або група об’єктів володіють.

Відповідно, як було зазначено вище, на відміну від сирих указників, що не містять жодної інформації про те, хто саме відповідає за виділену пам’ять, інтелектуальні указники моделюють володіння явно. Така модель дозволяє точно знати, коли об’єкт буде знищено, і запобігає конфліктам та невизначеності при одночасному використанні ресурсу з кількох місць у коді. Завдяки цьому розробник може мислити категоріями об’єктів і їх життєвих циклів, а не лише маніпулювати адресами в пам’яті.

Із переліку моделей володіння доречно виділити три основних та найбільш вживаних реалізації:

- Унікальне (unique) володіння – лише один указник володіє об’єктом у кожен момент часу. Це дозволяє забезпечити ефективне переміщення ресурсів без копіювання (натомість послуговуючись пересувною семантикою) і виключає як явище можливість подвійного вивільнення.
- Спільне (shared) володіння – кілька указників можуть одночасно володіти об’єктом. Безпека такої моделі володіння зазвичай

реалізовується за допомогою лічильника активних посилань на ресурс.

- Слабке (weak) володіння – указник не володіє об’єктом, а лише спостерігає за ним, не впливаючи на його життєвий цикл. Ця модель володіння використовується для запобігання утворення циклічних залежностей між об’єктами, якщо ті зберігають посилання одне на одного.

## 4.2. std::unique_ptr

Найпростішим та найефективнішим із точки зору накладних витрат на обчислювальні потужності та пам’ять у стандартній бібліотеці C++ STL є std::unique_ptr [14]. Цей указник, як загалом є очевидно з назви, реалізує модель унікального володіння ресурсом. При його знищенні, цей указник гарантує єдиний виклик деструктора, а отже безпечне звільнення пам’яті ресурсу, на який він показує. Відповідно, головна мета існування std::unique_ptr полягає в забезпеченні безпечного і максимально ефективного способу володіння динамічним ресурсом без зайвих накладних витрат. Через це, у багатьох ситуаціях std::unique_ptr є прямою заміною сирого указника, але з набагато вищим рівнем безпеки.

Окрім простої та ефективної абстракції, std::unique_ptr має вельми простий механізм роботи. Як вже було згадано, std::unique_ptr володіє ресурсом виключно, а отже його копіювання завжди заборонено, щоб запобігти ситуації, коли два указники намагаються отримати володіння над одним і тим самим ресурсом. Натомість дозволено переміщення: передача ресурсу від одного std::unique_ptr до іншого. Після переміщення вихідний укразник стає порожнім, тобто втрачає володіння об’єктом, а новий – перебирає на себе повне володіння.

Цей механізм роботи призводить до великої переваги порівняно із класичними (сирими) указниками – std::unique_ptr можна абсолютно безпечно

повертати у якості результату із функції, адже ресурс, яким володіє указник, буде передано у місце виклику функції, а потім – автоматично звільнено, після виходу об’єкта-указника за межі блоку видимості.

Щодо зазначеної вище високої продуктивності std::unique_ptr, то вона досягається простотою імплементації та забороною операцій по типу копіювання – екземпляри цього класу інтелектуальних указників (у основних реалізаціях C++ компіляторів) ніколи не перевищують у розмірі звичайний сирий указник. Це досягається за рахунок відсутності копіювання, використання простого деструктора і виключно локального керування ресурсом. У більшості реалізацій std::unique_ptr складається лише з одного поля – сирого указника на об’єкт. Завдяки цьому std::unique_ptr ідеально підходить для ресурсів, які не потрібно розділяти між кількома компонентами програми, і для яких критичною є висока продуктивність та безпека.

Розглянемо елементарний приклад створення та застосування std::unique_ptr до екземпляру структури координат у двовимірному просторі (Лістинг 8). Бачимо, що створення указника відбувається через std::make_unique – спеціальний шаблон функції, рекомендований до використання стандартною бібліотекою для інстанціонування нового указника-екземпляру. Уся подальша робота із інтелектуальним указником фактично аналогічна інтерфейсу взаємодії із сирим указником – екземпляри класу std::unique_ptr мають довизначені оператори розіменування та отримання доступу до полів об’єкту, а для випадків, коли програміст бажає отримати внутрішній сирий указник назад із інтелектуального, стандартна бібліотека надає метод get.

*Лістинг 8*

```
#include <memory>
#include <iostream>

struct Coordinate
{
    const int _x;
    const int _y;
};
```

```
static std::ostream &operator<<(std::ostream &os, const Coordinate &coord)
{
    return os << "Coordinate{x=" << coord._x << ", y=" << coord._y << '}';
}

int main()
{
    std::unique_ptr<Coordinate> ptr(std::make_unique<Coordinate>(5, 3));

    std::cout << ptr->_x << std::endl;
    std::cout << ptr->_y << std::endl;
    std::cout << *ptr << std::endl;
    return 0;
}
```

Наостанок варто зазначити, що std::unique_ptr підтримує користувацькі деструктори: програміст може передати власну функцію або об’єкт-функтор, який буде використано для знищення об’єкта та звільнення ресурсу. Це дозволяє використовувати std::unique_ptr для керування не лише динамічно виділеною пам’яттю, але й іншими ресурсами – наприклад, файловими дескрипторами, сокетами тощо.

## 4.3. std::shared_ptr та std::weak_ptr

Наступною імплементацією інтелектуального указника, яку має сенс розглянути, є std::shared_ptr – указник, що реалізує модель спільного володіння ресурсом [15]. Це означає, що сам ресурс, на який вказує один або група екземплярів std::shared_ptr, буде знищений лише тоді, коли останній указник з групи буде знищено або скинутий. Ця поведінка реалізується за допомогою внутрішнього лічильника посилань, який автоматично відстежує кількість активних власників ресурсу.

Головною перевагою std::shared_ptr є зручність застосування – він дозволяє ділитися об’єктом між різними частинами програми без необхідності явно управляти його життєвим циклом. Це особливо корисно у випадках, коли об’єкт використовується кількома компонентами одночасно – наприклад, у багатопоточному коді або callback-функціях. Іншим помітним плюсом є інтерфейс взаємодії із указником, що є схожим та, у певних аспектах функціонування ідентичним до сирих та унікальних указників, окрім декількох

специфічних методів на кшталт use_count (що надає інформацію про кількість активних посилань на ресурс – поточне значення лічильника спілки) та перейменованих функцій інстанціонування, як от std::make_shared замість std::make_unique.

Водночас, попри ці переваги, std::shared_ptr, а саме його модель володіння, має ряд відчутних недоліків. По-перше, кожен об’єкт std::shared_ptr зберігає додаткову службову інформацію, як мінімум – лічильник посилань. Це означає, що std::shared_ptr має більші накладні витрати порівняно зі std::unique_ptr, як у плані пам’яті, так і часу виконання. По-друге, семантика спільного володіння ускладнює контроль над точним моментом знищення об’єкта, і хоча це навряд чи призведе до невизначеної поведінки чи аварійної зупинки програми, така реалізація може банально не задовольняти вимоги особливо високопродуктивних чи критично обмежених у ресурсах систем.

Іншою серйозною проблемою std::shared_ptr є можливість створення циклічних залежностей між об’єктами – ситуацій, коли декілька об’єктів std::shared_ptr посилаються один на одного, утворюючи цикл та назавжди зберігаючи хоча б одне посилання на кожен ресурс, таким чином блокуючи виклик деструкторів, що зрештою призводить до витоку пам’яті.

Для вирішення та запобігання цієї проблеми стандартна бібліотека передбачила спеціальний, третій тип інтелектуального указника, що реалізує модель слабкого володіння ресурсом – std::weak_ptr [16]. Дотримуючись правил, визначених у своїй моделі, std::weak_ptr не володіє ресурсом на який посилається, а лише спостерігає за станом об’єкту, яким володіє певний std::shared_ptr та перевіряє його валідність.

Оскільки він не володіє об’єктом, використання std::weak_ptr не впливає на лічильник активних посилань групи std::shared_ptr, що й дозволяє розірвати цикл залежностей – у випадку, коли останній указник std::shared_ptr буде знищено, лічильник посилань знизиться до нуля, що у свою чергу спричинить

автоматичний виклик деструктора ресурсу. Цей процес відбудеться коректно та повністю незалежно від кількості std::weak_ptr, що вказують на ресурс, адже їхні посилання не є владними.

Наостанок варто зазначити, що для випадків, коли програміст хоче отримати владне посилання на об’єкт на який посилається std::weak_ptr, стандартна бібліотека C++ STL надає метод lock, що загалом має дві важливих особливості. По-перше, тип результату цього методу – std::shared_ptr, що має сенс, адже std::weak_ptr ніколи не може володіти об’єктом. По друге, значення його результату може бути або nullptr, що означає, що об’єкт, над яким програміст намагається отримати володіння вже було знищено, або ж валідним std::shared_ptr на ресурс, із яким програміст надалі може працювати так само, як із будь-яким іншим указником цього типу.

## 4.4. Особливості роботи з інтелектуальними указниками STL у багатопотоковому середовищі

Як вже було згадано наприкінці попереднього розділу, керування пам’яттю у багатопотокових застосунках набуває особливої складності через необхідність координованого одночасного доступу кількох потоків до спільних ресурсів. Тоді ми прийшли до висновку, що сирі указники та відсилки погано пристосовані до використання у багатопоточності, адже вони не мають жодних вбудованих механізмів які б полегшили цю інтеграцію та натомість перекладають усю відповідальність за забезпечення коректної синхронізації на розробника програми.

Безсумнівно, завжди існують випадки, коли саме такий тип взаємодії є бажаним. Гарним прикладом цьому слугують, наприклад, комп’ютерні ігри чи мікроконтролери – тобто програми, де оптимальне використання пам’яті та найвища можлива швидкість виконання є абсолютно критичними вимогами, а отже навіть тривіальні абстракції можуть більше заважати, аніж допомагати.

Тим не менш, очевидно, що переважна більшість програм, написаних на C++ не мають таких безкомпромісних обмежень, а використання програмістами сирих указників та відсилок при їх розробці майже завжди буде невиправданим. Причиною цьому є існування інтелектуальних указників, що, залежно від типу, здатні або забезпечити невимушену та безпечну взаємодію між потоками, або гарантувати абсолютну унікальність та ізоляцію інтерфейсу доступу до ресурсу від усіх інших потоків.

Почнемо із std::unique_ptr, що імплементує модель унікального володіння об’єктом та, забороняє будь-які копіювальні операції над собою. У багатопотоковому середовищі std::unique_ptr демонструє свою ефективність саме завдяки своїй простоті – він не потребує синхронізації доступу до контрольних структур, оскільки не має їх. Це означає, що у випадках, коли об'єкт не має бути доступним для кількох потоків одночасно, std::unique_ptr є не лише безпечним, але й оптимальним вибором.

Інакше поводиться std::shared_ptr, який призначений саме для тих ситуацій, де ресурс повинен бути доступним у декількох потоках одночасно. Найбільшою перевагою std::shared_ptr у багатопоточному середовищі є автоматична потокобезпечність його внутрішнього лічильника посилань, що зазвичай реалізується через std::atomic від певного цілочисельного типу. Операції копіювання, переміщення та видалення std::shared_ptr також синхронізовані на рівні реалізації стандартної бібліотеки, завдяки чому можна гарантувати, що ресурс буде знищений лише тоді, коли завершено його використання у всіх потоках, що мали на нього посилання. Але важливо пам’ятати, що така “автоматична” потокобезпечність std::shared_ptr розповсюджується виключно на дані його контрольної структури, насамперед лічильник посилань. Усі інші операції із цим типом указника, як от читання чи модифікація значення об’єкту, на який вказує указник, не є потокобезпечними та потребують окремої синхронізації, найчастіше на рівні структури даних, через такі інструменти як std::mutex та std::atomic.

# 5. Стратегії як інструмент проектування поведінки

## 5.1. Основні положення стратегій

У процесі створення програмного забезпечення розробник постійно стикається з необхідністю враховувати мінливість поведінки об’єктів. Поведінка, яка в одному випадку є сталою, в іншому потребує адаптації до умов, що змінюються під час виконання програми або у результаті налаштувань користувача. Часто такі зміни реалізуються шляхом додавання умовних операторів (if, switch) або дублювання коду в різних частинах програми. Це призводить до порушення принципу відкритості/закритості (Open/Closed Principle), уперше запровадженого Бертраном Меєром [17] та згодом популяризованого Робертом Мартіном [18]. Відповідно до цього принципу, код перестає бути гнучким і легко підтримуваним коли кожне нове уточнення або варіант поведінки потребує зміни вже існуючих компонентів.

Ця проблема ускладнюється ще більше у випадках, коли різні об’єкти або компоненти системи повинні реалізовувати схожі алгоритми, але з відмінностями в деталях. Пряме копіювання зміненої логіки або створення жорстко прив’язаних підкласів швидко призводить до надмірної зв’язаності коду, обмежує повторне використання, а також ускладнює тестування й розширення системи.

Відповідно, у таких ситуаціях виникає потреба у виділенні поведінки як окремого, змінного компоненту об’єкта. Саме цю ідею й реалізують стратегії. Отже, на концептуальному рівні, стратегія – це засіб, що дозволяє інкапсулювати певний алгоритм чи поведінку як окрему сутність, буквально як клас чи шаблон, який можна підставляти, змінювати або комбінувати без зміни коду класу, що використовує цю поведінку. Іншими словами – це механізм, який дозволяє делегувати реалізацію поведінки об’єкта іншому об’єкту із метою узагальнення та централізованого налаштування цієї поведінки.

Цікаво також, що реалізувати стратегію можна по-різному. Загалом можна виділити два найвідоміших, але докорінно різних підходи:

- Реалізація через поліморфні об’єкти, так звана динамічна стратегія, що здатна налаштовувати поведінку об’єкту під час виконання програми.
- Реалізація з використанням шаблонів та інших інструментів метапрограмування C++, так звана стратегія часу компіляції.

## 5.2. Динамічні стратегії

Поняття стратегій як способу інкапсуляції змінної поведінки було вперше систематично представлено у всесвітньовідомій книзі “Банди чотирьох” [19]. У ній, патерн “стратегія” описано як поведінковий взірець проектування, що дозволяє визначити сімейство алгоритмів, інкапсулювати кожен з них і зробити їх взаємозамінними. Сам підхід імплементації стратегії передбачає делегування реалізації змінної поведінки зовнішнім об’єктам, що підпорядковуються спільному інтерфейсу.

Імплементація динамічної стратегії, запропонована “Бандою чотирьох” проста – об'єкт, що володіє поведінкою, так званий контекст, зберігає посилання на інтерфейс стратегії, але нічого не знає про її конкретну реалізацію (Лістинг 9). Це дозволяє динамічно змінювати поведінку під час виконання програми, використовуючи різні об’єкти-стратегії, що реалізують спільний інтерфейс.

*Лістинг 9*

```
class SortContext
{
public:
    void setStrategy(std::unique_ptr<SortStrategy> newStrategy)
    {
        _strategy = std::move(newStrategy);
    }

    void executeSort() const
    {
        if (_strategy)
            _strategy->sort();
```

```
    }

private:
    std::unique_ptr<SortStrategy> _strategy;
};
```

Перевагою цього підходу є висока гнучкість. Зміна поведінки не потребує зміни контексту, а лише підміни конкретної реалізації стратегії. Це дає змогу легко розширювати систему новими алгоритмами, не змінюючи наявний код. Крім того, динамічні стратегії добре поєднуються з іншими шаблонами, наприклад, з фабриками чи декораторами – для створення ще більш адаптивних архітектур.

Однак цей метод імплементації стратегій має і певні недоліки. По-перше, динамічні стратегії потребують виділення об'єктів у динамічній пам’яті, що додає накладні витрати і створює потенційну необхідність у ретельному керуванні ресурсами. По-друге, використання віртуальних викликів призводить до, можливо й незначної на перший погляд, однак з часом вельми відчутної втрати продуктивності на кожному виклику методу стратегії. По-третє, динамічні стратегії не дозволяють здійснювати перевірку конкретної реалізації на етапі компіляції, що відкриває можливість помилок, які виявляються лише під час виконання програми.

## 5.3. Стратегії за Александреску

Ідея стратегій часу компіляції беззаперечно здобула розголосу та популярності у першу чергу завдяки книзі Александреску [20]. У ній, автор представив концепцію policy-based design, яка передбачає побудову поведінки об’єктів шляхом передачі шаблонних параметрів, що реалізують відповідну логіку. Таким чином, стратегія вибирається не під час виконання, як у класичному підході “Банди чотирьох”, а на етапі компіляції, що відкриває нові горизонти для ефективності, безпеки та гнучкості програми.

Розглянемо приклад реалізації стратегії часу компіляції (Лістинг 10). Бачимо, що така стратегія є спеціалізацією шаблону класу, що реалізує певну

змінну поведінку. Відповідно, замість зберігання вказівника на базовий клас, як це відбувається у динамічній версії, поведінка реалізується у вигляді шаблонного параметра, який передається у клас-контекст. Саме цей параметр реалізує необхідний алгоритм і контекст делегує йому виконання поведінки.

*Лістинг 10*

```
struct MergeSortPolicy
{
    static inline void sort()
    {
        std::cout << "Using MergeSort (compile-time)" << std::endl;
    }
};

template <typename SortPolicy>
class SortContext
{
public:
    void executeSort() const
    {
        SortPolicy::sort();
    }
};
```

Цікавим розгалуженням цієї реалізації, про яке варто згадати, є імплементація у якій сама стратегія також є шаблоном та залежить, наприклад, від певного типу T. У такому випадку, шаблон контексту можна розширити, так, що він прийматиме вже 2 параметри – власне тип T та шаблон стратегії, який буде інстанціоновано у реалізації контексту. Лістинг 11 демонструє приклад імплементації контексту у такому сценарії.

*Лістинг 11*

```
template <
    typename T,
    template <typename> typename SortPolicy
>
class SortContext
{
public:
    void executeSort() const
    {
        SortPolicy<T>::sort();
    }
};
```

Важливо також зазначити, що використання стратегій Александреску дозволяє компілятору проаналізувати усю структуру програми під час компіляції, оптимізувати виклики, а також виявити широкий ряд помилок ще

до запуску програми. Також, однією з ключових ідей Александреску є використання таких стратегій як гнучких будівельних блоків поведінки об’єктів програми, які можна комбінувати, наслідувати та налаштовувати залежно від потреб проекту.

Серед переваг цієї імплементації стратегій варто перш за все відзначити надзвичайно високу продуктивність. Оскільки вся поведінка визначається під час компіляції, компілятор має змогу виконати додаткові оптимізації. Крім того, компілятор перевіряє відповідність стратегій ще на етапі компіляції, що забезпечує сильну типову безпеку і зменшує ризик помилок на етапі виконання. Також варто згадати про гнучкість композиції поведінки – стратегії можуть вельми легко комбінуватися між собою, дозволяючи створювати нові варіації без дублювання коду. Це робить їх надзвичайно корисними у великих бібліотеках, фреймворках та складних шаблонних системах.

До беззаперечних недоліків можна віднести хіба що певне зростання складності коду, особливо у великих проектах. Шаблонні конструкції можуть бути дійсно важкими для читання, особливо для менш досвідчених розробників. Решта ж проблем мають менш значний характер, як от довший час компіляції, однак несправедливо було б вважати це вадою архітектури, адже кінцевий користувач продукту ніколи цього не відчує.

# 6. Демонстраційна програма “Архіватор”

Рішення щодо розробки архіватора як демонстраційного застосунку для цієї роботи насамперед вмотивоване тим, що задача архівації та розархівації даних (lossless compression/decompression) є досить складною з інженерної точки зору і вимагає ретельного підходу до управління пам’яттю, ефективної організації коду, а також забезпечення продуктивності при роботі з великими обсягами даних. Відповідно, програма-архіватор є надзвичайно вдалим та логічним контекстом для застосування як інтелектуальних указників, так і багатопоточності.

Насамперед, ознайомимось зі структурою проекту. На найвищому рівні (рівні директорії проекту) програма поділяється структурно на два компоненти – файл main.cpp та директорію app. Причина такого поділу проста – він наочно відділяє файл запуску програми від коду імплементації, бо файл запуску не реалізує логіку архіватора, а лише компонує надані кодом у директорії app складові, об’єднуючи їх у єдину програму.

Йдучи далі у директорію app маємо подальший розподіл на 4 директорії:

- app/config – відповідає за статичні налаштування архіватора та містить лише один файл config.h із набором декількох таких налаштувань-констант.
- app/core – містить усю основну логіку та код для архівації.
- app/model – містить класи-моделі, що об’єднують (інкапсулюють) групи даних програми у єдиному місці, надаючи інтуїтивні методи для взаємодії із цими даними.
- app/utils – містить файли програми, що оголошують та визначають перелік функцій/утиліт, що використовуються у решті компонентів додатку.

Розглянемо деякі з компонентів детальніше, починаючи з класу Token з моделі програми (Лістинг 12). Він описує трійку значень – offset, length, next_char, що позначає стиснений токен у обраному мною алгоритмі LZ77. Оскільки саме токени формують основний зміст стисненого потоку даних, цей клас імплементовано максимально легко та ефективно.

*Лістинг 12*

```
class Token final
{
public:
    struct HashFunctor final
    {
        inline std::size_t operator()(const Token &token) const
        {
            ...
        }
    };

public:
    Token() :
        _offset(0), _length(0), _next_char(0)
    {
    }

    Token(uint16_t offset, uint8_t length, uint8_t next_char) :
        _offset(offset), _length(length), _next_char(next_char)
    {
    }

    void write_to(std::vector<uint8_t> &output) const;
    static Token read_from(const std::vector<uint8_t> &input, const size_t pos);

    inline uint16_t offset() const { return _offset; }
    inline uint8_t length() const { return _length; }
    inline uint8_t next_char() const { return _next_char; }

private:
    uint16_t _offset;
    uint8_t _length;
    uint8_t _next_char;
};
```

Наступним класом, який варто розглянути, є TokenPool – реалізація взірця проектування Object Pool для токенів (Лістинг 13). Причина існування цього класу вельми проста та очевидна – навіть попри їх простоту та малий розмір, створення декількох однакових токенів точно не є оптимальним варіантом. Тут на допомогу і приходить TokenPool – клас-одинак, що зберігає кеш усіх попередньо створених токенів, імплементований за допомогою невпорядкованої (хеш) мапи. Він використовує std::mutex для

потокобезпечного знаходження чи створення токенів та std::shared_ptr для їх збереження, надаючи користувачеві спільне володіння при виклику метода get_token.

*Лістинг 13*

```
class TokenPool final
{
public:
    TokenPool(const TokenPool&) = delete;
    TokenPool &operator=(const TokenPool&)& = delete;

    static TokenPool &instance();
    std::shared_ptr<Token> get_token(const uint16_t offset,
                                     const uint8_t length,
                                     const uint8_t next_char);

private:
    TokenPool() = default;

private:
    std::unordered_map<
        Token,
        std::shared_ptr<Token>,
        Token::HashFunctor
    > _pool;
    std::mutex _mutex;
};
```

На відміну від файлів у директорії моделей проекту, реалізація ядра архіватора (у директорії app/core) не містить класів, натомість спираючись на процедурну парадигму програмування, а саме – використання функцій. Так, файл core.cpp реалізує такі функції як compress та decompress, що виконують однопоточну архівацію та розархівацію файлів і compress_parallel та decompress_parallel, що, відповідно до їх назви виконують ті ж операції у багатопотоковому середовищі. Водночас, у файлі присутні дві допоміжні функції compress_stream та decompress_stream, які абстрагують спільну поведінку згаданих вище функцій та використовуються як у однопоточних рішеннях (через простий послідовний виклик відповідно до правил потоку керування), так і у багатопоточних рішеннях (Лістинг 14).

*Лістинг 14*

```
futures.push_back(std::async(
    std::launch::async,
    // Compress the data within a block
    [block_data]() -> std::vector<uint8_t> { return compress_stream(block_data); }
));
```

# 7. Висновок

У процесі написання курсової роботи було здійснено комплексне дослідження засобів побудови потокобезпечного програмного забезпечення мовою C++. Основна увага приділялася аналізу механізмів багатопоточності, управління пам’яттю та застосуванню шаблонів проектування. У результаті виконаних завдань було отримано низку важливих теоретичних і практичних висновків.

По-перше, систематизовано знання про реалізацію багатопоточності у C++ та засоби керування потоками. Розглянуто відмінності між апаратною та симульованою багатопоточністю, визначено обмеження контекстного перемикання, а також акцентовано увагу на складності підтримання передбачуваного потоку керування у багатопотоковому середовищі.

По-друге, досліджено механізми синхронізації, зокрема примітиви std::mutex, std::lock_guard та std::atomic. Показано, що застосування RAII дозволяє суттєво знизити ризики помилок, пов’язаних із неправильним звільненням м’ютексів, а атомарні змінні у певних випадках забезпечують суттєві переваги в продуктивності без втрати потокобезпечності.

По-третє, розглянуто структуру пам’яті програм – стек і купу – та визначено основні проблеми управління пам’яттю у багатопотокових застосунках. Особливу увагу приділено інтелектуальним указникам std::unique_ptr, std::shared_ptr та std::weak_ptr як безпечним та ефективним засобам роботи з ресурсами. Продемонстровано, як моделі володіння допомагають уникнути витоків пам’яті, подвійного вивільнення та інших типових помилок.

Крім того, досліджено взірець проектування “стратегія” у двох варіантах реалізації — як динамічну поведінкову конструкцію за допомогою поліморфізму, так і як статичну конструкцію часу компіляції через шаблони.

Показано, що підхід policy-based design дозволяє підвищити гнучкість і ефективність архітектури програм, особливо у високопродуктивних системах.

Практичним результатом дослідження стало створення демонстраційної програми-архіватора, яка поєднує у своїй реалізації розглянуті у цій роботі концепції.

У рамках подальших досліджень доцільно розглянути використання більш розвинених моделей паралелізму, таких як actor model або корутини з C++20. Перспективним також є впровадження статичного аналізу потокобезпечності та дослідження підходів до автоматичного виявлення й запобігання станам гонки.

# 8. Список використаних джерел


1. Williams A. C++ Concurrency in Action: Practical Multithreading. Manning, 2012. 528 с.
2. std::thread – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/thread
3. std::reference_wrapper – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper
4. std::future – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/future
5. std::promise – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/promise
6. std::packaged_task – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/packaged_task
7. std::function – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/utility/functional/function
8. std::move – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/utility/move
9. std::async – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/async
10. std::mutex – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/mutex
11. std::lock_guard – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread/lock_guard
12. std::atomic – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/atomic/atomic
13. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014. 334 с.
14. std::unique_ptr – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/memory/unique_ptr

15. std::shared_ptr – cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/memory/shared_ptr
16. std::weak_ptr — cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/memory/weak_ptr
17. Meyer B. Object-Oriented Software Construction. Prentice Hall, 1988. 1280 c.
18. Martin R. C. Design Principles and Design Patterns. Object Mentor Inc., 2000. 34 c.
19. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 c.
20. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001. 352 c.