№ 11(52)
2025
НАУКА і ТЕХНІКА СЬОГОДНІ
серії: право, економіка, педагогіка, техніка, фізико-математичні науки

УДК 004.43

https://doi.org/10.52058/2786-6025-2025-11(52)-2885-2899

Шевцова Наталія Вікторівна кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0002-3401-5468

Кирик Тетяна Анатоліївна старший викладач кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0009-0002-2514-2534

Бабич Степанія Михайлівна кандидат технічних наук, доцент, доцент кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0003-2145-6392

Павлова Наталія Степанівна доктор педагогічних наук, доцент, професор кафедри цифрових технологій та методики навчання інформатики Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0002-7817-6781

ФУНКЦІОНАЛЬНІ КОНЦЕПЦІЇ У СУЧАСНІЙ МОВІ ПРОГРАМУВАННЯ C++

Анотація. Сучасний розвиток програмних систем характеризується зростанням потреби в абстрактних і модульних підходах до побудови програмного забезпечення. Однією з найактуальніших тенденцій останніх років є інтеграція функціональних концепцій у традиційні імперативні мови програмування, зокрема в C++.

Використання функціональних концепцій у C++ не лише розширює виразні можливості мови, а й сприяє підвищенню надійності, модульності та масштабованості програмних систем. Інтеграція лямбда-виразів, бібліотеки Ranges та монадичних операцій для std::optional створює в C++ потужне середовище для застосування функціональних підходів. Розробник отримує доступ до таких сучасних інструментів як композиція функцій, ліниві обчислення та безпечна обробка відсутніх значень. Ці можливості реалізуються без втрати низькорівневого контролю, що залишається фундаментальною перевагою мови C++.

Метою статті є комплексний аналіз функціональних концепцій у сучасному C++, дослідження їх теоретичних основ та практичної реалізації.

Особлива увага в статті приділяється монадичним патернам обробки даних, принципам композиції функцій та механізмам побудови декларативних конвеєрів обчислень за допомогою сучасних інструментів мови. На практичних прикладах реалізації алгоритмів обробки даних продемонстровано ефективність використання монадичних операцій та композиції функцій для створення читабельного та безпечного коду.

Аналіз демонструє, як поєднання функціональних та імперативних парадигм відкриває нові можливості для розробки високопродуктивних систем, безпосередньо впливаючи на покращення якості коду, підвищення тестованості та полегшення підтримки великомасштабних проєктів. Отримані результати підтверджують тенденцію до еволюції C++ у напрямку універсальної мови, здатної ефективно інтегрувати переваги різних парадигм програмування для створення сучасного програмного забезпечення.

Ключові слова: функціональне програмування, сучасні стандарти C++, лямбда-вирази, функції вищого порядку, композиція функцій, монадичні патерни.

Shevtsova Nataliia Viktorivna PhD, Associate Professor of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0002-3401-5468

Kyryk Tetiana Anatoliivna Senior Lecturer of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0009-0002-2514-2534

Babych Stepaniia Mykhailivna PhD, Associate Professor, Associate Professor of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0003-2145-6392

Pavlova Nataliia Stepanivna Doctor of Pedagogic Sciences, Associate Professor, Professor of the Department of Digital technologies and methods of teaching computer science, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0002-7817-6781

FUNCTIONAL CONCEPTS IN THE MODERN C++ PROGRAMMING LANGUAGE

Abstract. The evolution of modern software systems is marked by a growing demand for abstract and modular software design approaches. One of the most significant trends in recent years has been the incorporation of functional

programming concepts into traditional imperative languages, notably C++. Adopting functional concepts in C++ not only enhances the language's expressiveness but also improves the reliability, modularity, and scalability of software systems. The integration of lambda expressions, the Ranges library, and monadic operations for std::optional has established a robust foundation for functional programming within C++. This provides developers with modern tools such as function composition, lazy evaluation, and safe handling of missing values, all achieved without forfeiting the low-level control that is a cornerstone of C++.

This article aims to present a comprehensive analysis of functional concepts in modern C++, examining their theoretical underpinnings and practical applications. It places particular emphasis on monadic patterns for data processing, function composition principles, and the construction of declarative computation pipelines using contemporary language features. Through practical examples of data processing algorithms, the article demonstrates the efficacy of monadic operations and function composition in producing readable and robust code.

The analysis illustrates how the fusion of functional and imperative paradigms unlocks new potential for building high-performance systems, directly contributing to improved code quality, enhanced testability, and simplified maintenance of large-scale projects. These findings affirm the ongoing evolution of C++ into a versatile language, adept at integrating the strengths of diverse programming paradigms to meet the demands of modern software development.

Keywords: functional programming, modern C++ standards, lambda expressions, higher-order functions, function composition, monadic patterns.

Постановка проблеми. Аналіз сучасних підходів до розробки програмного забезпечення дозволяє визначити дві домінуючі парадигми програмування, що пропонують різні методології проєктування алгоритмів та архітектури програм. Вони ґрунтуються на різних світоглядних концепціях та математичних теоріях, що визначають принципи обчислень.

Імперативна парадигма реалізує модель обчислень, засновану на послідовній модифікації стану програми. Формально, імперативна програма може бути представлена як послідовність операторів, де кожен оператор здійснює трансформацію стану обчислювальної системи. Ця парадигма базується на математичній моделі обчислень Тюрінга через детерміновану зміну конфігурації машини.

Представники імперативної парадигми:

- мови структурного програмування: C, Pascal, Ada;
- об'єктно-орієнтовані мови: C++, C #, Java;
- скриптові мови: Python, JavaScript, PHP;
- системні мови: Assembler, Fortran, Go.

Імперативні мови програмування характеризуються явним керуванням станом, послідовною семантикою виконання та процедурною абстракцією.

Декларативна парадигма реалізує альтернативну модель обчислень, засновану на специфікації властивостей результату, тоді як імперативна парадигма зосереджується на алгоритмі його отримання. Формально, декларативна програма виражається як система рівнянь або логічних тверджень, що описують відношення між даними. Ця парадигма ґрунтується на математичній логіці та λ-численні, реалізуючи модель обчислень Черча.

Представники декларативної парадигми:

- функціональні мови: Haskell, Scheme;
- логічні мови: Prolog, Mercury, Datalog;
- мови специфікацій: SQL, HTML, CSS;
- функціонально-орієнтовані мови: Scala, Elixir, F#.

Декларативні системи характеризуються референтною прозорістю, композиційністю та відсутністю стану.

Функціональне програмування становить підмножину декларативної парадигми, що спеціалізується на обчисленнях через композицію функцій. Теоретичною основою функціонального програмування є λ-числення та теорія категорій. Прикладами мов програмування, що реалізують цей підхід, є Haskell і PureScript.

Ключовими характеристиками функціонального підходу є чистота функцій, незмінність даних, розгляд функцій як об’єктів першого класу та композиційність. Функціональне програмування дозволяє будувати програми шляхом комбінування математичних функцій, що забезпечує високий рівень абстракції, модульності та верифікованості коду.

Еволюційний розвиток мови C++ відображає фундаментальну трансформацію у підході до програмування, де традиційна імперативна модель поступово збагачується елементами функціонального програмування.

Починаючи з революційного стандарту C++11, спостерігається систематичне впровадження функціональних концепцій, що перетворює C++ з виключно імперативної мови на мультипарадигмову платформу.

Ця конвергенція парадигм є відповіддю на сучасні вимоги до розробки складних програмних систем, де поєднання низькорівневого контролю імперативного підходу з високорівневими абстракціями функціонального програмування створює потужний симбіоз. Стандарт C++11 ввів ключові функціональні елементи, такі як лямбда-вирази, автоматичне виведення типів (type inference) та розумні вказівники, що започаткували нову еру у розвитку мови.

Подальші стандарти (C++14, C++17, C++20, C++23) поглибили цю тенденцію, додавши складніші функціональні конструкції, включаючи алгебраїчні типи даних, pattern matching, розширену підтримку шаблонного

метапрограмування та бібліотеку ranges. Цей напрямок розвитку демонструє свідомий вибір комітету з стандартизації C++ у напрямку інтеграції перевірених часом функціональних концепцій у рамки традиційної імперативної мови.

Така конвергенція дозволяє розробникам поєднувати переваги обох парадигм: ефективність та низькорівневий контроль імперативного підходу з безпекою типів, композиційністю та виразністю функціонального програмування. Це робить сучасний C++ особливо привабливим для розробки високопродуктивних систем, де критично важливі як швидкодія, так і надійність коду.

Завдяки цьому сучасний C++ стає потужним засобом для створення високопродуктивних систем, що потребують водночас високої швидкодії та надійності коду.

Аналіз останніх досліджень і публікацій виявляє низку пріоритетних напрямів у дослідженні поєднання функціонального та імперативного підходів у мові програмування C++.

Теоретико-методологічний напрям досліджує фундаментальні основи інтеграції функціональних концепцій. Роботи B. Stroustrup та C. Scalfani аналізують філософію та методологічні переваги такого синтезу, зокрема композиційність, детермінованість та безпеку типів [1, 2]. *Практико-орієнтований напрям* зосереджений на прикладних аспектах використання функціональних інструментів. Дослідження S. Meyers та публікації щодо монадичних операцій у C++23 демонструють конкретні методики застосування лямбда-виразів, функцій вищого порядку та шаблонів обробки помилок [3, 4].

Стандартизаційний напрям відображає формальне впровадження функціональних концепцій через специфікації ISO/IEC 14882. Особливу увагу приділено еволюції від C++11 до C++23, де послідовно посилюється підтримка функціональних парадигм [5, 6, 7].

Дидактичний напрям представлений інноваційними підходами до викладення функціональних концепцій через практичні аналоги та приклади, зокрема в працях M. Plachta [8].

Сучасні дослідження демонструють консенсус щодо переваг гібридного підходу, однак залишаються невирішеними питання оптимізації продуктивності та ефективного поєднання парадигм у високонавантажених системах.

Мета статті: аналіз функціональних концепцій у сучасному C++, дослідження їх теоретичних основ і практичної реалізації, а також оцінка їх впливу на архітектуру програмних систем.

Виклад основного матеріалу. Функціональне програмування становить фундаментальний підхід до конструювання програмних систем, який ґрунтується на принципах математичної теорії функцій. Історично ця парадигма бере початок від λ-числення Алонзо Черча, яке було розроблено як

формальна система для вивчення функцій та їх властивостей. Важливо розуміти, що ФП – це не просто набір технічних прийомів, а цілісна філософія програмування, яка радикально змінює спосіб мислення розробника.

Функціональне програмування ґрунтується на системі взаємопов'язаних принципів, що формують його методологічну основу та визначають специфіку програмної архітектури [8]:

Принцип чистих функцій та його наслідки. Чисті функції становлять концептуальний фундамент функціонального програмування. Чиста функція визначається двома ключовими властивостями: детермінованістю та відсутністю побічних ефектів. Детермінованість означає, що для фіксованого набору вхідних аргументів функція завжди повертає однаковий результат, незалежно від зовнішніх умов або стану системи. Відсутність побічних ефектів гарантує, що виконання функції не впливає на зовнішнє середовище – не змінює глобальні змінні, не виконує операції введення-виведення, не модифікує вхідні параметри.

Ці властивості мають глибокі практичні наслідки. Детермінованість робить поведінку програми передбачуваною та піддається формальному аналізу. Відсутність побічних ефектів усуває складні проблеми синхронізації в багатопоточному середовищі, оскільки відпадає необхідність в управлінні спільним станом. Крім того, чисті функції демонструють властивість референтної прозорості – будь-який виклик функції можна замінити на її результат без зміни семантики програми.

Незмінність даних та її архітектурні переваги. Концепція незмінності даних тісно пов'язана з принципом чистих функцій. Замість модифікації існуючих структур даних, функціональний підхід передбачає створення нових на основі старих. Ця, на перший погляд, неефективна стратегія на практиці призводить до значного спрощення архітектури програмних систем.

Незмінність усуває цілий клас помилок, пов'язаних з неочікуваною зміною стану. У традиційних імперативних системах складні взаємодії між компонентами часто призводять до важковиявних помилок, коли один модуль неочікувано змінює дані, які використовуються іншим модулем.

Функціональний підхід елімінує цю проблему, гарантуючи, що дані, передані в функцію, залишаються незмінними.

Архітектурні переваги незмінності особливо очевидні в контексті паралельного програмування. Оскільки дані не змінюються, зникає потреба у складних механізмах блокування для забезпечення узгодженості стану, що суттєво спрощує розробку конкурентних систем і підвищує їхню надійність.

Функції як об'єкти першого класу та композиційність. У функціональному програмуванні функції розглядаються як об'єкти першого класу, що означає рівноправність функцій з іншими типами даних. Функції можуть

передаватися як аргументи, повертатися як результати, зберігатися в структурах даних. Ця властивість відкриває можливість для створення функцій вищого порядку – функцій, які приймають інші функції як параметри або повертають їх як результат.

Композиція функцій є потужним засобом абстракції, що дозволяє будувати складні програми з простих компонентів. Принцип композиційності полягає в тому, що складність системи має визначатися складністю її компонентів та способом їх поєднання, а не додатковими емерджентними властивостями. Такий підхід дозволяє створювати програмні системи, які піддаються модульному аналізу та верифікації.

Рекурсія замість циклів. Відмова від циклічних конструкцій на користь рекурсії у функціональному програмуванні ґрунтується на фундаментальних принципах теорії рекурсивних функцій. Традиційні циклічні конструкції суперечать принципу незмінності стану та чистоти функцій, оскільки вимагають явної модифікації стану лічильників та створення побічних ефектів у вигляді зміни значень змінних. На противагу цьому, рекурсія безпосередньо відображає математичну практику визначення функцій через рекурентні співвідношення, що робить її природнішою для функціональної парадигми.

Рекурсивний підхід забезпечує значні переваги у композиційності та модульності коду, оскільки функції природно піддаються декомпозиції на базовий випадок та рекурсивний крок. Крім того, при належній оптимізації рекурсивні рішення досягають аналогічної з ітеративними підходами продуктивності, забезпечуючи при цьому кращу читабельність, простішу паралелізацію та більш високу ступінь абстракції.

Система типів та гарантії коректності. Сучасні функціональні мови програмування часто включають потужні системи типів, які забезпечують додаткові гарантії коректності на етапі компіляції. Системи типів, засновані на теорії Хіндлі-Мілнера, дозволяють автоматично виводити типи виразів, зменшуючи кількість анотацій, які повинен надавати розробник.

Алгебраїчні типи даних забезпечують структурований спосіб визначення складних структур даних. Типи-суми дозволяють представляти значення, яке може належати до одного з декількох варіантів, а типи-добутки – комбінувати кілька значень в одну структуру. Разом вони утворюють потужний засіб моделювання предметної області.

Система типів також підтримує поліморфізм, що дозволяє створювати узагальнені алгоритми, які здатні працювати з різними типами даних. Параметричний поліморфізм забезпечує безпеку типів при збереженні гнучкості узагальненого програмування.

Практичні аспекти та ефективність. Незважаючи на теоретичні переваги, функціональне програмування стикається з практичними викликами,

особливо в області ефективності. Незмінність даних може призводити до додаткових витрат пам'яті та часу виконання. Однак сучасні техніки, такі як персистентні структури даних та оптимізація спільного використання пам'яті, дають змогу значно зменшити ці витрати.

Ліниві обчислення представляють інший важливий аспект функціонального програмування. Замість негайного обчислення всіх виразів, функціональні мови відкладають обчислення до моменту, коли результат дійсно потрібен. Це дозволяє створювати потенційно нескінченні структури даних та оптимізувати обчислення шляхом усунення непотрібних обчислень.

Вплив на архітектуру програмних систем. Функціональне програмування має глибокий вплив на архітектуру програмних систем. Принципи модульності, композиційності та незмінності сприяють створенню систем з низькою зв'язністю та високою зчепленістю. Особливу значимість ці принципи набувають у контексті паралельних та розподілених систем, де відсутність спільного стану та гарантована детермінованість чистих функцій усувають необхідність у складних механізмах синхронізації. Незмінність структур даних гарантує безпеку при одночасному доступі з багатьох потоків виконання або розподілених вузлів, тоді як композиційна природа функціонального коду спрощує розподіл обчислень між різними компонентами системи. Такі системи не лише легше тестувати, модифікувати та розширювати, але й вони природно адаптуються до сучасних вимог масштабованості та обробки даних у реальному часі, що робить функціональні підходи особливо привабливими для розробки високонавантажених та розподілених застосунків.

Функціональні концепції в сучасному C++ сприяють упровадженню архітектурних патернів, що значно знижують зв'язність компонентів системи та підвищують коефіцієнт повторного використання коду. Ці патерни базуються на фундаментальних принципах функціонального програмування та реалізуються через механізми сучасної мови. Серед найбільш виразних представників цих патернів слід виділити наступні архітектурні моделі, які знайшли широке застосування в сучасній практиці програмування на C++.

Конвеєр обробки даних (Pipeline Pattern) реалізується через інтеграцію бібліотеки ranges та функцій вищого порядку, таких як transform, filter та reduce. Цей підхід забезпечує можливість композиції декількох обчислень у єдиний декларативний ланцюг трансформацій, що виключає побічні ефекти та гарантує детермінованість виконання.

Незмінні структури даних (Immutable Data Structures) реалізуються через механізми константності, семантику переміщення та спеціалізовані шаблонні класи. Цей підхід усуває ризики виникнення стану гонки при паралельному доступі, спрощує процес тестування компонентів системи та забезпечує передбачуваність поведінки програми.

Функції вищого порядку (Higher-Order Functions) реалізуються через систему шаблонів, лямбда-вирази та спеціалізовані типи. Вони надають можливість створення узагальнених алгоритмів, які можуть бути багаторазово використані в різних контекстах програми, підвищуючи рівень абстракції та спрощуючи композицію програмних компонентів.

Монадичні патерни (Monadic Patterns) реалізуються через спеціалізовані типи-контейнери, такі як std::optional, std::expected, std::future та власні шаблонні реалізації. Ці патерни надають структурований підхід до обробки потенційно відсутніх значень, асинхронних обчислень та помилкових станів, інкапсулюючи логіку обробки побічних ефектів у композиційний ланцюг операцій. Вони дозволяють уникнути глибоко вкладених умовних конструкцій та спрощують обробку виняткових ситуацій, забезпечуючи більш лінійний та зрозумілий потік виконання програми.

Ці теоретичні концепції знаходять практичне втілення в сучасних стандартах C++ (C++11-C++23), які послідовно інтегрують функціональні парадигми в традиційно імперативну модель мови, поєднуючи переваги обох підходів. Важливою інновацією виступили лямбда-вирази, що дозволяють створювати анонімні функції з доступом до змінних із зовнішньої області видимості. Вони можуть передаватися як аргументи та повертатися як результати інших функцій, що безпосередньо втілює концепцію функцій як об'єктів першого класу.

Синтаксис лямбда-виразів [5]

[захоплення](параметри) -> тип_результату { тіло }

дозволяє точно контролювати семантику захоплення змінних через різні моделі:

[=] для іммутабельного захоплення за значенням;

[&] для мутабельного захоплення за посиланням;

[x, &y] змішані варіанти для гібридного підходу.

Особливістю сучасних стандартів є підтримка ініціалізацій у захопленні, наприклад [value = compute_value()], що дозволяє створювати ефективні та безпечні замикання. Наприклад, конструкція

```
auto square = [](int x) { return x * x; }; // оголошення
// виклик лямбда-функції
int result = square(5); // result = 25
```

створює чисту функцію, яка може передаватися як об'єкт першого класу, а більш складна лямбда-функція ілюструє можливість ініціалізації стану без необхідності явного конструктора.

```
auto multiplier = [factor = 2](int x) { return x * factor; };
// виклик лямбда-функції
int result = multiplier(5); // result = 10
```

Ключове слово auto, яке зустрічається в розглянутих прикладах, у сучасному C++ реалізує механізм автоматичного виведення типу на етапі компіляції, що значно спрощує роботу зі складними типами даних та покращує читабельність коду. Замість явного вказування типу змінної, компілятор самостійно визначає його на основі ініціалізатора, що особливо корисно при роботі з шаблонними типами, ітераторами та лямбда-виразами. Цей механізм забезпечує типобезпеку, усуваючи неявні приведення типів, та сприяє створенню універсального коду, адаптованого до майбутніх змін у структурі даних. Використання auto стало особливо актуальним із розвитком стандартної бібліотеки C++, де типи, що повертаються складними шаблонними виразами, часто є громіздкими для ручного специфікування.

Поява лямбда-виразів та автоматичного виведення типів у стандарті мови C++ безпосередньо відкриває шлях до реалізації функцій вищого порядку, які є фундаментом функціонального підходу. Композиція функцій реалізується через шаблонні конструкції:

```
auto compose = [](auto f, auto g)
{ return [f, g](auto x) { return f(g(x)); }; };
```

Така композиція відповідає математичній операції $(f \circ g)(x) = f(g(x))$ і дозволяє будувати складні трансформації даних із простих компонентів. Наприклад:

```
auto double_value = [](auto x) { return x * 2; };
auto to_string = [](auto x) { return std::to_string(x); };
// Композиція: to_string(double_value(x))
auto double_and_stringify = compose(to_string, double_value);
//виклик композиції функцій
auto result = double_and_stringify(42); // "84"
```

На практиці ці механізми інтегруються з бібліотекою ranges C++20. Бібліотека реалізує принцип відкладених обчислень, де трансформації застосовуються лише під час ітерації, що оптимізує використання пам'яті та обчислювальних ресурсів. Композиційна модель дозволяє будувати складні конвеєри обробки даних через пайп-оператор (|), що формально відповідає математичній композиції функцій.

Фундаментальну теоретичну основу бібліотеки ranges складає система взаємопов'язаних концепцій, що формалізують роботу з послідовностями даних. Центральною концепцією виступає range – формальне визначення послідовності елементів із чітко заданими початком і кінцем, що узагальнює традиційні пари ітераторів. Ця абстракція забезпечує уніфікований інтерфейс для різноманітних структур даних, від стандартних контейнерів до потоків введення-виведення. Важливим компонентом системи є концепція view – спеціалізований тип range, який дозволяє будувати ланцюги обробки даних без

їх копіювання, застосовуючи перетворення лише в момент звернення до елементів. Вона реалізує модель відкладених обчислень, де перетворення застосовуються лише за необхідності під час ітерації. Ця властивість забезпечує ефективну композиційність операцій та оптимізацію використання пам'яті.

Завершує теоретичний фундамент система адаптованих алгоритмів – узагальнених версій класичних STL (Standard Template Library) алгоритмів, що оперують безпосередньо з range-об'єктами.

Ці алгоритми забезпечують повну типобезпеку та сумісність з функціональним підходом, дозволяючи будувати складні конвеєри обробки даних із гарантією коректності на етапі компіляції. Інтеграція цих трьох компонентів створює строгу математичну основу для функціональної обробки послідовностей у сучасному C++.

Формальна семантика композиції у бібліотеці ranges реалізується через перевантаження пайп-оператора, що дозволяє будувати алгебраїчну структуру операцій над діапазонами згідно синтаксису:

```
namespace std::views {
 template<typename R, typename P>
 auto operator|(R&& range, P&& predicate) -> range-type; }
```

Розглянемо приклад, який демонструє, як операції над даними виражаються через функціональні трансформації замість імперативних циклів.

```
std::vector<int> data = {1, 2, 3, 4, 5};
auto result = data | std::views::transform([](int x) { return x * x; })
 | std::views::filter([](int x) { return x % 2 == 0; });
// Виклик ланцюга обробки: ітерація призводить до
//обчислення фільтрації та трансформації
 for (int value : result)
 std::cout << value << " "; // виведе: 4 16
```

За допомогою filter і transform можна комбінувати обчислення без явного управління ітераторами та проміжними контейнерами, що відображає принцип композиційності функцій у чистому функціональному програмуванні.

Принцип композиції, реалізований у бібліотеці ranges для послідовностей даних, знаходить своє природне продовження в монадичних патернах. Останні узагальнюють цей підхід для роботи з обчисленнями, що характеризуються специфічною семантикою виконання, такою як потенційна відсутність результату, можливість помилки чи асинхронність виконання. У C++ така додаткова семантика інкапсулюється за допомогою шаблонних типів, серед яких [6]:

- std::optional – представляє обчислення, яке може не мати результату;
- std::expected – моделює обчислення, що може завершитися помилкою;
- std::future – описує асинхронне обчислення.

Монадичні патерни формалізують спосіб компонувати подібні обчислення в єдиний ланцюг, зберігаючи при цьому їх семантику та забезпечуючи типобезпеку. Монада визначається як алгебраїчна структура, що складається з трьох компонентів: типового конструктора та двох операцій unit і bind, які задовольняють монадичні закони.

Типовий конструктор у C++ реалізується через шаблонні класи, де параметр шаблону визначає тип значення, що інкапсулюється:

```
template<typename T>
class Monad
{ // Реалізація специфічної семантики виконання };
```

Операція unit реалізує відображення T → M<T>, трансформуючи звичайне значення в екземпляр монадичного типу.

```
template<typename T>
Monad<T> unit(T value);
```

На практиці unit реалізується через конструктори відповідних типів:

```
std::optional<int> opt = 10; // unit для std::optional
std::expected<int, Error> exp = 404; // unit для std::expected
```

Операція bind реалізує відображення M<T> × (T → M<U>) → M<U> та забезпечує композицію монадичних обчислень:

```
template<typename T, typename F>
auto bind(Monad<T> m, F f) -> decltype(f(m.extract()));
```

Коректність монадичної структури визначається трьома законами:

1. ліва ідентичність:

$$bind(unit(x), f) \equiv f(x);$$

2. права ідентичність:

$$bind(m, unit) \equiv m;$$

3. асоціативність:

$$bind(bind(m, f), g) \equiv\equiv bind(m,[\&](auto\ x)\{return\ bind(f(x),g);\}).$$

Стандарт C++23 впроваджує нові монадичні операції для типу std::optional, що реалізують теоретичні принципи композиції обчислень з контекстом [4, 7].

Формальне визначення монади як трійки (M, unit, bind) знаходить практичне втілення через операції and_then, transform та or_else. Ці інструменти дозволяють будувати композиційні конвеєри обробки даних із автоматичною обробкою потенційно відсутніх значень, усуваючи необхідність у явних перевірках стану.

Таким чином, C++23 інтегрує сучасні функціональні парадигми у мову системного програмування, зберігаючи переваги продуктивності та типобезпеки.

Операція and_then реалізує фундаментальну монадичну операцію bind. На практиці це виражається через композицію функцій, що повертають std::optional:

```
std::optional<int> safe_divide(int a, int b)
{ return (b == 0) ? std::nullopt : std::optional(a / b);}
std::optional<int> add_five(int x)
{ return x + 5; }
std::optional<int> square(int x)
{ return x * x;}
// Ланцюг монадичних операцій
auto result = std::optional{100}
 .and_then([](int x) { return safe_divide(x, 2); }) // 100/2=50
 .and_then([](int x) { return safe_divide(x, 5); }) // 50/5=10
 .and_then(add_five) // 10+5=15
 .and_then(square) // 15*15=225
 .and_then([](int x) { return safe_divide(x, 15); }); // 225/15=15
// Результат: std::optional{15}
```

Ланцюжок операцій переривається, коли будь-яка операція and_then повертає std::nullopt.

Операція transform реалізує відображення: (T → U) → (M<T> → M<U>). Відмінність між and_then та transform полягає в типі функції, яку вони виконують: and_then приймає функції, що повертають std::optional і використовується для ланцюгів операцій з можливими помилками, тоді як transform працює з функціями, що повертають звичайні значення, які автоматично обгортаються в std::optional.

```
auto result = std::optional{5}.transform([](int x)
{ return x * x; }); // Результат: std::optional{25}
```

Операція or_else забезпечує обробку альтернативних сценаріїв для порожнього стану монади:

```
std::optional<int> getFromCache()
{ return std::nullopt; } // порожнє значення для прикладу
std::optional<int> getFromDatabase()
{ return 100; } // альтернативне значення
auto value = getFromCache()
 .or_else(getFromDatabase); //результат 100
```

Монадичний підхід дозволяє компонувати прості операції в єдині ланцюги обробки даних, автоматично вирішуючи такі типові проблеми, як помилки виконання чи відсутність значень. На практиці монадичні патерни застосовують для обробки помилок без використання винятків, асинхронних обчислень, модульного аналізу структурованих даних та керування станом

програми. Ця парадигма забезпечує строгу математичну основу для створення надійних, композиційних програмних систем у C++.

Висновки. Сучасний C++ демонструє поєднання різних парадигм програмування, де функціональні підходи стають невід'ємною частиною традиційної імперативної моделі мови. Послідовне доповнення C++ такими інструментами, як лямбда-вирази, автоматичне виведення типів, бібліотека Ranges та монадичні операції для std::optional, перетворює функціональне програмування з теоретичної концепції на практичний інструмент для створення надійного, ефективного та якісного програмного забезпечення.

Функціональні патерни в C++ забезпечують ряд ключових переваг, включаючи покращену композиційність, ізоляцію побічних ефектів, статичну типобезпеку та зручність моделювання складних обчислювальних процесів.

Вони ефективно застосовуються для обробки помилок без винятків, асинхронних обчислень, аналізу структурованих даних та управління станом.

Запровадження цих патернів стимулює розробників до декларативного стилю мислення, формуючи модульні та чисті обчислювальні блоки, які легко піддаються композиції та багаторазовому використанню.

Це сприяє створенню більш надійних, масштабованих і легких у супроводі програмних систем.

Література:

1. Stroustrup B. A Tour of C++. 2nd ed. Addison-Wesley, 2018. 240 p.
2. Scalfani Ch. Why Functional Programming Should Be the Future of Software Development (IEEE Spectrum, 2022) URL: https://spectrum.ieee.org/functional-programming (date of access: 14.11.2025).
3. How to Use Monadic Operations for std::optional in C++23. URL: https://www.cppstories.com/2023/monadic-optional-ops-cpp23/ (date of access: 14.11.2025).
4. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014. 334 p.
5. ISO/IEC 14882:2011. Programming languages — C++ (C++11 Standard). 3rd ed. Geneva : ISO, 2011. 1358 p.
6. ISO/IEC 14882:2020. Programming Languages — C++ (C++20 Standard). 6th ed. Geneva : ISO, 2020. 1836 p.
7. ISO/IEC 14882:2024. Programming Languages — C++ (C++23/C++2024 Standard). 7th ed. Geneva : ISO, 2024. 1954 p.
8. Plachta M. Grokking Functional Programming. 1st ed. Manning Publications, 2023.450p.

References:

1. Stroustrup, B. (2018). *A tour of C++* (2nd ed.). Addison-Wesley.
2. Scalfani, C. (2022). Why functional programming should be the future of software development. *IEEE Spectrum*. Retrieved November 14, 2025, from https://spectrum.ieee.org/functional-programming
3. Meyers, S. (2014). *Effective modern C++: 42 specific ways to improve your use of C++11 and C++14*. O'Reilly Media.

4. How to use monadic operations for std::optional in C++23. (2023). Retrieved November 14, 2025, from https://www.cppstories.com/2023/monadic-optional-ops-cpp23/
5. ISO/IEC 14882:2011. (2011). *Programming languages — C++* (3rd ed.). ISO.
6. ISO/IEC 14882:2020. (2020). *Programming languages — C++* (6th ed.). ISO.
7. ISO/IEC 14882:2024. (2024). *Programming languages — C++* (7th ed.). ISO.
8. Plachta, M. (2023). *Grokking functional programming* (1st ed.). Manning Publications.