бездротового зв'язку, що може застосовуватись для комунікації на великих відстанях (до 50 км). Подібно до цього, технологія LoRa здатна забезпечувати стабільний зв'язок на відстані до 5 км [4]. Крім того, дослідники зосереджують увагу й на технологіях короткого радіусу дії, таких як Bluetooth, ZigBee, NFC. Ці технології розглядають у контексті комунікації транспортного засобу з пристроями, які розташовані всередині нього (V2D), а також з тими, що його оточують (V2P, V2I).

Незважаючи на велику кількість технологій, що пройшли польові випробування, і на те, що дослідження у сфері комунікації транспортних засобів тривають уже досить давно, ці технології досі не набули значного поширення. Це пояснюється багатьма факторами. Один із них полягає в тому, що для більшості технологій потрібні додаткові пристрої, які треба встановлювати як у транспортні засоби, так і в інфраструктуру. Наприклад, для технологій DSRC, C-V2X, LoRa необхідні бортові пристрої (OBUs) у кожному транспортному засобі для забезпечення комунікації типу V2V, а також додаткові стаціонарні пристрої вздовж дороги (RSUs) для взаємодії з інфраструктурою. Утім, такі рішення, у випадку з технологіями DSRC та LoRa, забезпечують лише децентралізовану взаємодію. Щоб отримати доступ до глобальної мережі, ці RSU необхідно підключити до Інтернету. В таких умовах потрібне дуже чітке узгодження дій між виробниками транспорту і муніципальними органами, адже кожна технологія має свою специфікацію. До того ж, різні технології можуть конфліктувати між собою: наприклад, у DSRC та C-V2X перетинаються частотні діапазони, що може створювати перешкоди при одночасній роботі. Інша проблема полягає в тому, що наукова спільнота наразі переважно зосереджена на розбудові децентралізованих мереж. Це загострює зазначені вище проблеми та породжує додаткові труднощі при проектуванні мереж і забезпеченні їхньої безпеки. Отже, зміна фокусу з повністю децентралізованої моделі на використання централізованої інфраструктури з децентралізованими елементами може докорінно змінити загальну ситуацію і прискорити розвиток усієї галузі інтелектуальних транспортних систем. У цьому контексті виникає потреба в проектуванні і розробці відкритої системи та універсального відкритого протоколу, здатних забезпечити єдиний механізм обміну повідомленнями та централізовані засоби безпеки. Це дозволить розробляти застосунки для ITS без прив'язки до конкретних технологій зв'язку чи виробників.

Список джерел: (до 5 джерел, оформлений за стандартом ДСТУ/ВАК)
[1] Kenney J. B. Dedicated Short-Range Communications (DSRC) Standards in the United States [Електронний ресурс] / John B. Kenney // Proceedings of the IEEE. — 2011. — Т. 99, № 7. — С. 1162–1182. – Режим доступу: https://doi.org/10.1109/jproc.2011.2132790 (дата звернення: 27.10.2025).
[2] Ott J. Drive-thru internet: IEEE 802.1 1b for "automobile" users [Електронний ресурс] / J. Ott, D. Kutscher // IEEE INFOCOM 2004, Hong Kong, PR China. – [Б. м.]. – Режим доступу: https://doi.org/10.1109/infcom.2004.1354509 (дата звернення: 27.10.2025).
[3] Wang X. An Overview of 3GPP Cellular Vehicle-to-Everything Standards [Електронний ресурс] / Xuyu Wang, Shiwen Mao, Michelle X. Gong // GetMobile: Mobile Computing and Communications. – 2017. – Т. 21, № 3. – С. 19–25. – Режим доступу: https://doi.org/10.1145/3161587.3161593 (дата звернення: 27.10.2025).
[4] Zadobrischi E. Enhancing Scalability of C-V2X and DSRC Vehicular Communication Protocols with LoRa 2.4 GHz in the Scenario of Urban Traffic Systems [Електронний ресурс] / Eduard Zadobrischi, Ștefan Havriliuc // Electronics. – 2024. – Т. 13, № 14. – С. 2845. – Режим доступу: https://doi.org/10.3390/electronics13142845 (дата звернення: 27.10.2025).

ДО ПИТАННЯ УЗАГАЛЬНЕНОГО МЕТАПРОГРАМУВАННЯ В C++ / TOWARDS THE ISSUE OF TEMPLATE METAPROGRAMMING IN C++

Бублик В. В., Трохимчук А. А. / Boublik V., Trokhymchuk A.
Національний університет "Києво-Могилянська Академія" кафедра мультимедійних систем / National University of Kyiv-Mohyla Academy
04655, Київ, вул. Григорія Сковороди, 2,
факультет інформатики,
E-mail: boublik@ukma.edu.ua, a.trokhymchuk@ukma.edu.ua

The paper explores C++ metaprogramming as a form of declarative functional programming that originates from Alonzo Church's lambda calculus. Paper describes the imperative approach based on iteration with the functional style centered around recursion. The core of

C++ metaprogramming, the template system, is analyzed. Key techniques such as *Substitution Failure Is Not An Error* (SFINAE) and template specializations are explored in detail. The methods enable compile-time computations, turning the template language into a Turing-complete functional programming language within C++ itself. As a practical demonstration, the paper presents a modern solution to the "Abstract Factory" design pattern problem, originally formulated by A. Alexandrescu, showcasing the power of template metaprogramming in C++.
Ключові слова: метапрограмування, мова шаблонів, C++, SFINAE.

Програмування історично розвивалося у двох основних стилях: імперативному та декларативному (зокрема, функціональному). Ітерація є ключовим засобом організації обчислень в імперативному програмуванні, яке, за своєю суттю, є програмуванням над комірками пам'яті. На противагу цьому, рекурсія є основним інструментом функціонального (декларативного) програмування, яке не вимагає явного керування пам'яттю та операцій присвоєння.

Мовою C++, яка традиційно розглядається як мова імперативного програмування, концепції функціонального стилю реалізовані через механізм метапрограмування шаблонами. Це так зване «статичне метапрограмування», де всі обчислювальні процеси відбуваються на етапі компіляції, до початку виконання основної програми. Такий підхід дозволяє значно зменшити навантаження на етапі виконання коду, переклавши обчислення на час компіляції. Власне, програма компілюється один раз, а виконується тисячі чи мільйони разів, тому такий підхід дозволяє зекономити значну кількість ресурсів.

Розглянемо класичну задачу обчислення найбільшого спільного дільника (НСД). Традиційна рекурсивна реалізація мовою C++ виконує обчислення під час роботи програми:

```
int gcd(const int m, const int n) { return (n == 0) ? m : gcd(n, m % n); }
```

Проте, використовуючи метапрограмування шаблонами, ці обчислення можна перенести на етап компіляції. Для цього створюється основний шаблон та його часткова спеціалізація, що слугує умовою завершення рекурсії:

```
// Основний шаблон
template<int M, int N> struct GCD {
   static consteval int gcd() {
      return GCD<N, M% N>::gcd();
   }
};
// Часткова спеціалізація для завершення рекурсії
template<int M> struct GCD<M, 0> {
   static consteval int gcd() { return M; }
};
```

У такому випадку виклик `int n = GCD<168, 392>::gcd()` змусить компілятор самостійно обчислити результат, і в об'єктний код потрапить лише готовий результат. Це і є суть статичного метапрограмування: програма, написана розробником, генерує і виконує кінцевий код на етапі компіляції, жодних накладних витрат під час виконання програми. В результаті мова шаблонів C++ перетворилася з простого засобу узагальненого програмування для створення контейнерів на повноцінну функціональну мову, повну за Тюрінгом [1], завдяки двом ключовим механізмам: SFINAE та спеціалізації.

SFINAE («*Substitution Failure Is Not An Error*») [2] — це правило, згідно з яким невдала підстановка шаблонного параметра не є помилкою компіляції. Замість цього відповідний шаблон просто виключається з розгляду. Це дозволяє створювати перевантажені функції, які компілятор обиратиме залежно від характеристик типів, що підставляються. Наприклад, можна створити два конструктори для класу рядків, один з яких буде обрано, якщо довжина рядка дозволяє розмістити його у внутрішньому буфері (реалізація *Small String Optimization*), а інший — якщо потрібно динамічно виділяти пам'ять. SFINAE є основою для реалізації умовної логіки на етапі компіляції.

Спеціалізація шаблонів — це механізм, що дозволяє перевизначити поведінку шаблону для конкретних типів або значень. Спеціалізація буває повною, коли надано всі шаблонні параметри (по суті, спеціалізація одного конкретного випадку), і частковою. Яскравим прикладом є std::vector<bool>, який є частковою спеціалізацією std::vector<T>. Замість того, щоб зберігати кожне булеве значення в окремому байті (що призвело б до 700% надлишкових витрат пам'яті), спеціалізована версія використовує бітові поля, зберігаючи 8 значень в одному байті.

Спеціалізація також відіграє роль умови зупинки в рекурсивних метафункціях. Розглянемо метафункцію для обчислення чисел Фібоначчі:

$$Fibonacci_n = Fibonacci_{n-1} + Fibonacci_{n-2}, n = 2,3,\ldots$$

Основний шаблон реалізує рекурсивне правило, а спеціалізації для 0 та 1 надають кінцеві значення для зупинки рекурсії.

```
template<int n> struct Fibonacci {
   static consteval int fib() {
      return Fibonacci<n - 1>::fib() + Fibonacci<n - 2>::fib();
   }
};
template<> struct Fibonacci<1> { static consteval int fib() { return 1; } };
template<> struct Fibonacci<0> { static consteval int fib() { return 0; } };
```

Ефективність таких обчислень можна підвищити за допомогою кешування, зберігаючи вже обчислені результати у статичних членах класу, що дозволяє оптимізувати алгоритмічну складність з експоненційної до лінійної.

Сила метапрограмування полягає у вирішенні реальних задач. У 2001 році Андрей Александреску [3] сформулював задачу реалізації шаблону проектування «*Абстрактна Фабрика*» з єдиним шаблонним методом Create<T>. На той час елегантне рішення було неможливим, і доводилося використовувати громіздкі макроси препроцесора. Сучасне метапрограмування дозволяє вирішити цю задачу за допомогою структури даних списку типів (type_list). Така фабрика може перевіряти на етапі компіляції, чи входить певний тип до списку продуктів, які вона може створювати, і чи можливо його сконструювати з наданих аргументів, використовуючи SFINAE.

```
type_list<Triangle, Rectangle, Circle> factory{};
Triangle* t = factory.create<Triangle>(3, 4, 5);
```

Такий підхід робить фабрику «дружньою до шаблонів» і дозволяє легко її розширювати, не змінюючи існуючий інтерфейс.

Таким чином, метапрограмування шаблонами в C++ виводить нас за рамки чисто імперативного програмування, дозволяючи виконувати складні обчислення на етапі компіляції, генерувати ефективний код та реалізовувати складні патерни проектування елегантно та безпечно з точки зору типів.

Список джерел:

1. Veldhuizen T. L. C++ Templates are Turing Complete, University of Indiana Technical Report, 2003.
2. Vandevoorde D. C++ Templates: The Complete Guide / David Vandevoorde and Nicolai M. Josuttis, Addison-Wesley, Boston, MA, 2002
3. Alexandrescu A. Modern C++ design: generic programming and design patterns applied / Andrei Alexandrescu. Addison-Wesley Professional, 2001. 352 p.

DATA STREAMING PIPELINE FOR THE QUADCOPTER FLIGHT CONTROL STACK

T. Zavaliy, N. Shakhovska, V. Iatsyshyn

Lviv Polytechnic National University

79000, Lviv, Kn. Romana str., 5, Department of Artificial Intelligence

E-mail: taras.i.zavalii@lpnu.ua, nataliya.b.shakhovska@lpnu.ua, volodymyr.p.yatsyshyn@lpnu.ua