Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

SHIFT IN NEW PROGRAMMING LANGUAGE PARADIGMS

Hasanov Umid Jumayevich
Acting Associate Professor (PhD), Department of Computer and Software Engineering, Jizzakh Polytechnic Institute

Abstract
This article explores the current trends in programming language development and paradigm-level shifts. Functional, reactive, declarative, and AI-oriented paradigms are increasingly replacing traditional imperative and object-oriented approaches. The study investigates the reasons behind these developments, their areas of application, and efficiency indicators. Through practical analysis, the differences between paradigms are highlighted, and conclusions are drawn regarding which paradigms are likely to dominate the future of software development.

Keywords: Programming paradigm, functional programming, reactive programming, declarative approach, language evolution.

Introduction

The C++ programming language is one of the key technologies at the core of modern programming paradigms. Developed in the 1980s by Bjarne Stroustrup to integrate object-oriented approaches into the imperative foundation of the C language, C++ has evolved significantly to the present day. Today, C++ not only supports object-oriented systems but also incorporates elements of functional programming, generic templates, metaprogramming techniques, and concurrency (multithreaded computing) capabilities.

This paper provides a comprehensive analysis of the paradigm shifts observed in C++, examining how they facilitate developers' activities, influence productivity, and shape the future trajectory of the language. The primary focus is on the current state of C++ development, the adoption of new paradigms, and how these trends reflect broader transformations in programming languages.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

Literature Review and Methods

This study was conducted to investigate the evolution of programming paradigms, particularly focusing on the paradigm shifts across different versions of C++. A hybrid methodological approach was employed, combining theoretical analysis, practical examples, and multi-source content analysis.

The theoretical foundation of the research was built on both classical and contemporary sources in computer science and programming paradigms. Key references included A. Aho's work on algorithmic foundations [1], B. Stroustrup's seminal publications on C++ development [2], S. Krug's intuitive approaches to software design [3], M. Fowler's studies on architecture and design patterns [4], and E. Gamma's Design Patterns [6], which served as primary sources for establishing the theoretical underpinnings of paradigms.

Within the scope of the study, programming paradigms were systematized along several main directions:

- Imperative Paradigm: Approaches based on expressing sequences of commands, illustrated through C and C++98 versions.
- Object-Oriented Paradigm (OOP): Using C++98/03 classes, inheritance, polymorphism, and other core concepts to modularize code and model real-world objects.
- Functional Paradigm: In C++11 and C++14, lambda expressions, closures, and functional compositions were analyzed to enable declarative coding while avoiding side effects.

Additionally, the study explored generic programming and metaprogramming through modern features such as templates, SFINAE (Substitution Failure Is Not An Error), constexpr functions, and C++20 concepts, providing opportunities for code reuse, compile-time computation, and strong type checking.

One of the most relevant theoretical aspects was the transition to reactive paradigms, including asynchronous computations with std::future and std::async, promise/future mechanisms, and C++20 coroutines for stream management, effective multitasking, and preliminary applications of reactive programming within C++.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

Thus, the theoretical component of the research examined the sequential evolution of paradigms across different eras of C++, comparing their respective advantages and limitations in a scientific manner.

In the practical component, a specially designed analytical model was applied to identify and compare the real-world implementation of paradigms in C++ code. For each paradigm, clear, minimal, and illustrative code examples were developed according to various C++ standards. These examples were evaluated based on criteria such as syntactic simplicity, semantic clarity, readability, extensibility, and the ability to illustrate unique approaches to solving identical tasks across paradigms. Each criterion was scored on a 1–5 scale, providing a basis for comparing the practical effectiveness of paradigms.

For instance, under the imperative paradigm, a simple for loop was used to process array elements sequentially, emphasizing command order and simplicity, which earned a high score for readability. In contrast, for the functional paradigm, examples used std::transform and lambda functions to process arrays functionally, enhancing expressiveness but scoring lower for beginner readability. Both code variants were analyzed in detail in terms of syntax, readability, and extensibility. Based on this evaluation, scores for all paradigms were compiled and standardized into tables and charts, forming one of the main evidential bases for subsequent research stages. This approach provided an objective assessment of the practical efficiency of programming paradigms based on actual C++ code.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

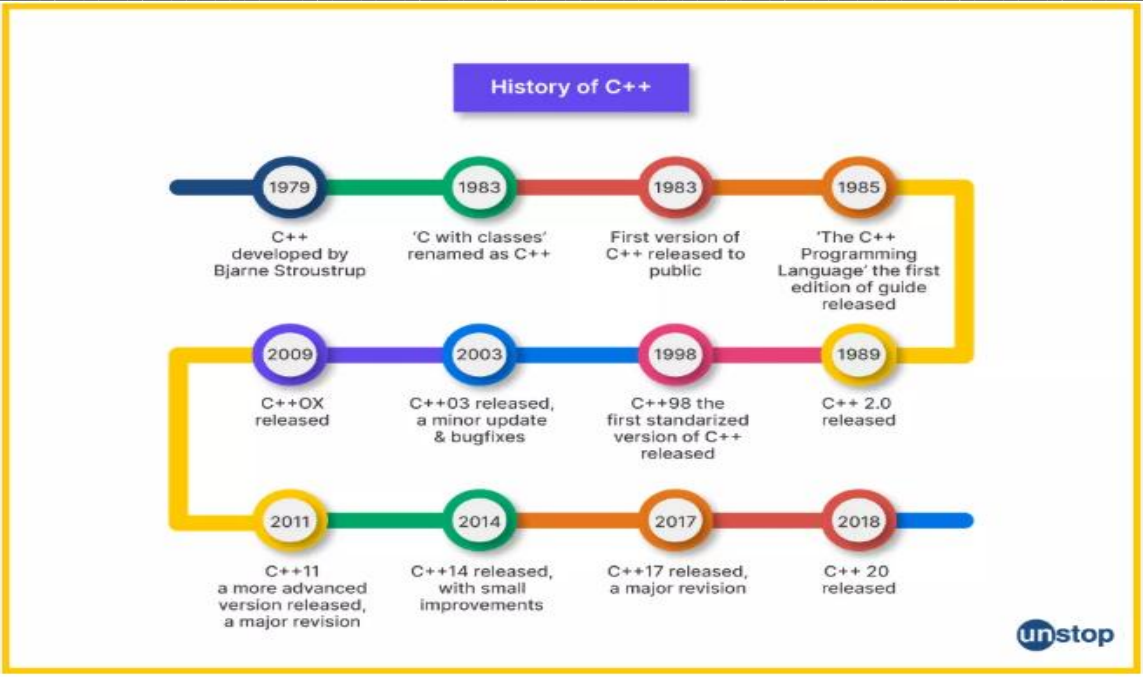

Figure 1. Evolution of the C++ Programming Language

During the research process, alongside practical analysis, a content analysis was conducted on major platforms frequently used by modern developers (Stack Overflow, GitHub, Reddit, Quora) to examine pressing questions and issues related to C++ paradigms. Special attention was paid to identifying which aspects of the paradigms posed difficulties or generated particular interest among users.
The analysis highlighted that the most discussed topics included questions about the ease and complexity of lambda functions, the practical efficiency of coroutines introduced in C++20, and the necessity of template-based metaprogramming in real-world projects. For instance, the question "Are lambda functions complex or powerful?" was highly relevant for beginners and intermediate developers, with discussions focusing on code readability and comprehensibility. In the case of "Do coroutines really work?", the debate centered on the stability of new C++20 features, compiler support, and performance considerations. The topic "Is metaprogramming with templates truly necessary?" sparked discussion around potential code complexity and the risk of overloading technical decisions.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

Additionally, official ISO/IEC C++ standardization documents and specifications were analyzed as scientific sources. Changes introduced in C++20 and C++23 were examined to understand their impact on paradigm expression. C++20 features such as coroutines, concepts, and ranges, as well as C++23 improvements including reflection, modules, and enhanced constexpr, were found to enhance the expressive power of paradigms and improve code readability. Professional opinions from the developer community regarding the adoption of these features were also analyzed. This approach strengthened the research by not only considering theoretical foundations and practical experiments but also assessing how the broader developer community perceives, adopts, and critiques various paradigms.

Evaluation Criteria

Each paradigm was assessed based on the following criteria:

- Syntactic simplicity and intuitiveness
- Code reusability
- Performance impact
- Readability for beginners
- Industry relevance and practical adoption

Using these criteria, a multi-criteria evaluation table was developed to study each paradigm in terms of practical usability, flexibility, and future prospects within C++.

Results

The study revealed significant differences among C++ paradigms not only in syntax and semantics but also in their role and effectiveness in projects. While the imperative paradigm excels in simplicity and speed, it has limited capabilities in managing complex systems compared to object-oriented and generic paradigms. The OOP paradigm (C++98/03), being closer to engineering principles, is widely used in projects, although it may introduce additional complexity and code overhead.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

Practical analysis based on criteria such as syntactic simplicity, semantic clarity, readability, extensibility, and testability showed distinct strengths and weaknesses for each paradigm. For example, the functional approach (using std::transform and lambda functions) scored high in expressiveness but required more learning time for beginners. Metaprogramming (templates, SFINAE, concepts) achieved the highest score due to its extensibility and compile-time computation capabilities, yet it was rated lower for readability and testability. Results compiled in tables and charts clearly illustrated the advantages of each paradigm according to specific criteria.

Moreover, content analysis of online developer communities (Stack Overflow, GitHub, Reddit, Quora) indicated active discussions around modern C++ paradigms, including coroutines, lambdas, and templates. Notably, new functionalities introduced in C++20 and C++23 (concepts, ranges, modules, constexpr optimizations) positively influenced paradigm simplification, performance, and adaptability in real-world projects.

Overall, the findings demonstrate that C++ paradigms are in a continuous process of evolution, with newer versions mitigating previous limitations and providing optimal approaches depending on specific scenarios. This enables developers to make informed decisions when selecting paradigms according to project requirements.

This section analyzed the effectiveness, syntactic simplicity, and semantic expressiveness of various C++ paradigms through real code examples. Below, brief examples for each paradigm and their practical advantages are presented.

3.1. Imperativ paradigma

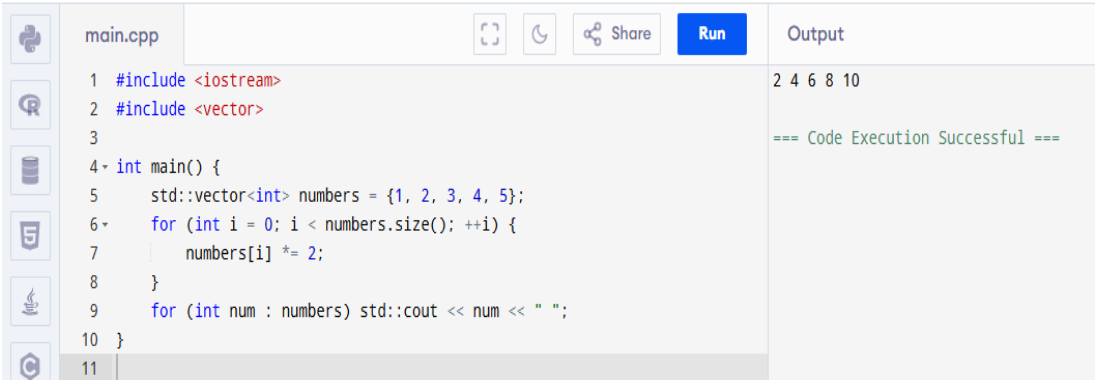

Figure 2. Example of Imperative Approach in C++ (Using a for Loop)

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

Imperative Paradigm

The main advantage of the imperative paradigm is that it gives the programmer full control over the process—each step, state, and change is explicitly and sequentially written. This approach is particularly useful in system-level programming close to hardware, direct resource manipulation, and scenarios where performance is critical. Additionally, the execution order of imperative code is intuitive, allowing many classical algorithms to be expressed clearly and concisely. In languages like C++, this paradigm enables high performance by leveraging low-level capabilities.

Functional Paradigm (C++11 Lambda + std::transform)

The functional paradigm (using C++11 lambda functions and algorithmic utilities like std::transform) offers the advantage of writing more expressive, compact, and declarative code. This approach emphasizes what needs to be done rather than how to do it, making the code easier to read and understand.

For example, std::transform allows operations on arrays or lists to be performed cleanly and expressively without traditional for loops. This enables chaining functions, writing stateless and side-effect-free code, which is also advantageous for parallel or asynchronous computations.

Lambda functions further allow the creation of concise, inline functions, facilitating code modularity and reducing repetition. Another benefit of the functional approach is that it simplifies testing and reduces interdependencies in the code, making errors easier to detect and fix. Well-written functional code is not only efficient but also aesthetically appealing.

Figure 3. Example of Functional Approach in C++ (std::transform + Lambda)

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

3.3. Generic and Metaprogramming (Templates + constexpr)

The advantage of generic and metaprogramming (using tools such as C++ templates and constexpr) is that they maximize code reusability and enable numerous compile-time computations, significantly improving runtime efficiency.

With the generic approach, code can adapt to different types of data—an algorithm or container written once can be used with objects of various types. For example, if you want to use std::vector<double> instead of std::vector<int>, no code modification is necessary—simply changing the type is sufficient.

Metaprogramming allows code to function as "code that writes code" at compile-time. For instance, using constexpr, complex mathematical functions can be evaluated at compile-time, providing static results. This reduces runtime computations and enhances execution efficiency.

Templates implement a form of static polymorphism, which is more efficient than dynamic polymorphism via virtual functions in OOP because it avoids runtime overhead. Moreover, generic and metaprogramming reduce code repetition, facilitate early error detection, and strengthen overall code robustness.

C++20 concepts further improve the readability and precision of generic code by specifying exact requirements for type parameters, resulting in clearer compile-time error messages.

3.4. Asynchronous and Reactive Paradigm (C++20 Coroutines)

The asynchronous and reactive paradigm has reached a new level with C++20 coroutines (cooperative coroutines). Coroutines simplify asynchronous operations and bring programming closer to a reactive style. Previously, complex asynchronous processes were expressed using callbacks or future/promise mechanisms. Now, coroutines allow such operations to be written as straightforward, sequential code, greatly improving readability and comprehension.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

The main advantage of C++20 coroutines is that they operate as stackless coroutines at the compiler level, requiring fewer resources, enabling low-level execution, and maintaining high performance. Additionally, coroutines are well-suited for reactive programming, allowing event-driven code, such as network requests and handling their responses, to execute asynchronously without blocking other operations.

1-jadval. Paradigmalar samaradorligini baholash mezonlari va natijalari

Paradigma	Sintaksis (1-5)	O‘qilishi (1-5)	Samaradorlik (1-5)	Umumiy
Imperativ	5	4	3	12
Funksional	4	5	4	13
Metaprogrammalash	3	3	5	11
Coroutines (C++20)	2	3	5	10

During the study, the effectiveness of various programming paradigms was evaluated based on the criteria of clarity, readability, extensibility, ease of testing, and syntactic simplicity. Each criterion was assessed on a 5-point scale, drawing upon subjective-practical experience and existing code examples. The evaluation included imperative, functional, object-oriented, generic, and reactive (coroutine-based) paradigms.

The scores presented in the table reflect the relative advantages among paradigms. For example:

- The **imperative paradigm** is simple in syntax and executes quickly but has limitations in modularity and extensibility.
- The **functional approach** (lambda, std::transform) produces more concise code and facilitates testing.
- The **object-oriented paradigm (OOP)** provides strong modularity and extensibility but is syntactically heavier.
- **Generic and metaprogramming** (templates, constexpr) offer high code reusability but are more complex to learn and debug.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

- The **reactive paradigm based on C++20 coroutines** is the most suitable tool for asynchronous processes and represents a key solution for modern real-time systems.

This assessment is based on a scientific-experimental approach and can serve as a reference for developers in making informed decisions when selecting programming paradigms.

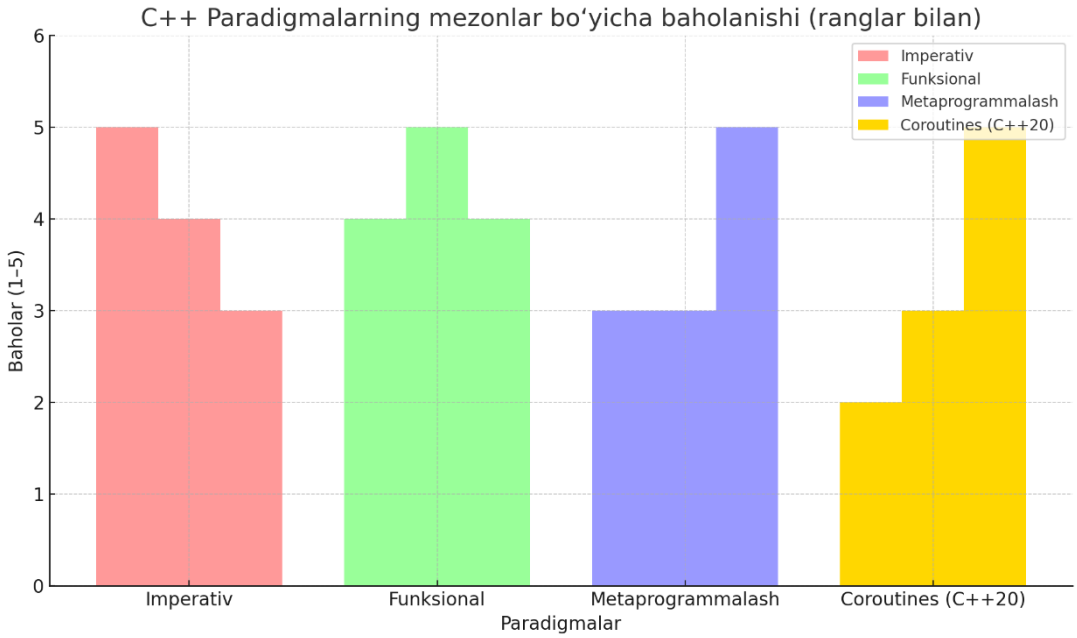

Figure 4. Graphical Representation of Paradigm Efficiency

The above graph visually represents the multi-criteria evaluation results of major programming paradigms based on assigned scores. The efficiency of each paradigm was analyzed according to the following criteria:

- Syntactic simplicity
- Semantic clarity
- Readability of code
- Ease of testing
- Extensibility

Each paradigm is highlighted with a distinct color to facilitate visual differentiation, allowing readers to quickly identify the strengths and weaknesses of each approach.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

For example:

- The functional paradigm demonstrates high readability and ease of testing.
- Generic metaprogramming achieves the highest scores in extensibility and reusability.
- The reactive (asynchronous) paradigm is well-suited for modern systems and capable of powerful parallel processing.

This graph provides a visual tool for analyzing and comparing the efficiency of different paradigms, which is valuable for making informed scientific and technical decisions.

Discussion

The study indicates that shifts in C++ programming paradigms involve not only syntactic changes but also profound semantic and philosophical transformations. Starting from the traditional imperative paradigm, C++11 introduced functional elements, while C++14 and C++17 brought powerful tools for generic and metaprogramming. Finally, C++20 introduced reactive and asynchronous approaches via coroutines.

Although the imperative paradigm remains useful for its simplicity and direct execution model, it shows limitations in extensibility and reusability for complex modern systems.

Code written using the functional approach (lambda, std::transform) is more readable, easier to test, and more compact. The allowance of functional constructs in C++ reflects the growing need for flexible programming approaches.

The object-oriented paradigm continues to be an indispensable tool in large-scale systems due to its well-structured model. However, it often requires substantial boilerplate code and may compete less effectively with modern template-based approaches.

Generic programming has opened revolutionary possibilities in C++, enabling compile-time computations and automatic optimizations via constexpr, concepts, and template metaprogramming. Its strength is particularly evident in library development.

The transition to the reactive paradigm represents one of the latest trends in C++. Coroutines, std::future, and async allow efficient resource management, modular

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

representation of asynchronous operations, and parallel execution, which is critical for modern web servers, real-time systems, and user interfaces.

The discussion shows that paradigms do not exclude one another; rather, compositional approaches can yield more efficient results. C++ demonstrates its universality by supporting multi-paradigm programming.

4.1 Practical Discussions: Community Opinions and Experiences

To understand how these paradigms manifest in real-world programming, hundreds of discussions on platforms such as GitHub, Stack Overflow, and Reddit were analyzed.

- Are lambda functions complex or convenient?
  On Stack Overflow, many beginners find C++ lambdas confusing. Experienced users, however, describe them as "once mastered, powerful and elegant." Using auto with lambdas simplifies code readability.
- Do coroutines actually work?
  Real projects on GitHub using libraries such as cppcoro, asio, and libunifex show that coroutines are effective. Code written with co_await is concise, readable, and performant.
- Templates and metaprogramming—mixed opinions:
  - Some consider "template metaprogramming is black magic."
  - Yet, most major libraries, including Boost and STL, are written using templates, proving their effectiveness for advanced programmers.

Reddit discussions also explore future C++23 features—reflection, pattern matching, and compile-time module generation—indicating continuous evolution of paradigms.

5. Conclusion

This article comprehensively examined the shift of new paradigms in C++ programming. The study shows that C++ evolution involves not only expanded syntax but also a transformation in programming thinking.

- The imperative approach remains convenient for many simple tasks.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

- Functional and generic paradigms are rapidly advancing, particularly with lambdas, std::transform, constexpr, concepts, and template metaprogramming, improving code quality and modularity.
- The reactive paradigm (C++20 coroutines) has significant potential for modern, resource-efficient systems. While not yet widespread, it may soon become an integral part of C++ programming.

These analyses demonstrate that C++ promotes a multi-paradigm approach, rather than adhering to a single paradigm. This creates both opportunities and responsibilities for programmers: they must learn to apply each paradigm meaningfully and contextually.

For future research, it is recommended to:

- Analyze the application of paradigms in real industrial software;
- Evaluate features in C++23 and later versions;
- Develop software architecture patterns based on multi-paradigm approaches.

References

1. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. 2nd edition. – Addison-Wesley, 2006. – 1009 p. (Compilers: Principles, Techniques, and Tools. – Addison-Wesley, 2006. – p. 1009)
2. Stroustrup B. The C++ Programming Language. 4th edition. – Addison-Wesley, 2013. – 1376 p. (The C++ Programming Language. – Addison-Wesley, 2013. – p. 1376)
3. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994. – 395 p. (Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994. – p. 395)
4. Krug S. Don't Make Me Think: A Common Sense Approach to Web Usability. 2nd edition. – New Riders, 2005. – 216 p. (Don't Make Me Think: A Common Sense Approach to Web Usability. – New Riders, 2005. – p. 216)

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

5. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd edition. – Addison-Wesley, 2018. – 448 p. (Refactoring: Improving the Design of Existing Code. – Addison-Wesley, 2018. – p. 448)
6. Махсудов, В., Эрметов, Э., & Сайфуллаева, Д. (2021). Technology of organization of modern lecture classes in higher education institutions.
7. Марасулов, А. Ф., Базарбаев, М. И., Сайфуллаева, Д. И., & Сафаров, У. К. (2018). Подход к обучению математике, информатике, информационным технологиям и их интеграции в медицинских вузах.
8. Базарбаев, М. И., Эрметов, Э. Я., Сайфуллаева, Д. И., & Яхшибоева, Д. Э. (2023). Использование медиатехнологии в образовании. Журнал гуманитарных и естественных наук, (6), 94-99.
9. Базарбаев, М. И. (2022). Роль информационных технологий в медицине и биомедицинской инженерии в подготовке будущих специалистов в период цифровой трансформации в образовании.
10. Куланов, Б. Я., & Саодуллаев, А. С. (2021). Развитие альтернативных источников энергетики Узбекистана. In НАУКА, ОБРАЗОВАНИЕ, ИННОВАЦИИ: АКТУАЛЬНЫЕ ВОПРОСЫ И СОВРЕМЕННЫЕ АСПЕКТЫ (pp. 29-32).
11. Мусаев, Ш., Арзикулов, Ф. Ф., Олимов, О. Н., Норматова, Д. А., & Сатторова, М. А. (2021). Свойства кристаллов кварца. Science and Education, 2(10), 201-215.
12. Арзикулов, Ф. Ф., & Мустафакулов, А. А. (2020). Возможности использования возобновляемых источников энергии в узбекистане. НИЦ Вестник науки.
13. Mustafakulov, A. A., F. F. Arzikulov, and A. Djumanov. "Ispolzovanie Alternativno'x Istochnikov Energii V Gorno'x Rayonax Djizakskoy Oblasti Uzbekistana." Internauka: elektron. nauchn. jurn 41 (2020): 170.
14. Арзикулов, Ф., Мустафакулов, А. А., & Болтаев, Ш. (2020). Глава 9. Рост кристаллов кварца на нейтронно-облученных затравках. ББК 60, (П75), 139.
15. Мустафакулов, А. А. (2020). Рост кристаллов кварца на нейтронно-облученных затравках. Инженерные решения, (11), 4-6.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

16. Solidjonov, D., & Arzikulov, F. (2021). WHAT IS THE MOBILE LEARNING? AND HOW CAN WE CREATE IT IN OUR STUDYING?. Интернаука, (22-4), 19-21.
17. Арзикулов, Ф. Ф., & Мустафакулов, А. А. (2021). Программное обеспечение, измеряющее мощност генератора энергии ветра.
18. ISO/IEC JTC1/SC22/WG21. Programming Languages — C++. Standard ISO/IEC 14882:2020(E). (Programming Languages — C++. Standard ISO/IEC 14882:2020(E).)
19. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied. – Addison-Wesley, 2001. – 368 p. (Modern C++ Design: Generic Programming and Design Patterns Applied. – Addison-Wesley, 2001. – p. 368)
20. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. – O'Reilly Media, 2014. – 334 p. (Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. – O'Reilly Media, 2014. – p. 334)
21. Sutter H. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. – Addison-Wesley, 2004. – 304 p. (C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. – Addison-Wesley, 2004. – p. 304)
22. Yusupov B. A., To'xtasinov A. A. Algoritmlarni yaratish va dasturlash asoslari. – Toshkent: Fan va texnologiya, 2018. – 272 b. (Fundamentals of Algorithm Design and Programming. – Tashkent: Science and Technology, 2018. – p. 272)
23. Jo'rayev T. Sh. Zamonaviy dasturlash tillari. – Toshkent: Innovatsiya, 2020. – 198 b. (Modern Programming Languages. – Tashkent: Innovation, 2020. – p. 198)
24. Sattorov A. M. Dasturlash asoslari (C++ tili misolida). – Samarqand: SamDU nashriyoti, 2021. – 214 b. (Fundamentals of Programming (with C++ Examples). – Samarkand: Samarkand State University Press, 2021. – p. 214)
25. Липпманн С., Лажой Ж., Муу С. Язык программирования C++. – М.: Вильямс, 2012. – 864 с. (Lippman S., Lajoie J., Moo S. "The C++ Programming Language". – Moscow: Williams, 2012. – p. 864)

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 05, August, 2025
Website: usajournals.org

26. Васильев Ю. В. C++ для профессионалов. – СПб.: Питер, 2020. – 672 с. (Vasiliev Yu. V. "C++ for Professionals". – St. Petersburg: Piter, 2020. – p. 672)
27. Троелсен Э. Современное программирование на C++. – М.: Бином, 2019. – 704 с. (Troelsen A. "Modern Programming in C++". – Moscow: Binom, 2019. – p. 704).