Universität Siegen
Seminar Paper

# The C++ Programming Language

Historical Development, Design Principles, and Practical Application

submitted by:
Khadija Azzouzi
1778812
Date: 09.01.2026

## Contents

## 1 Introduction

C++ is a programming language that is often discussed. It is described either as an outdated relic of systems programming or as a necessary tool for performance-critical software. Despite these polarized views, C++ keeps evolving and remains widely used in areas where efficiency, resource control, and long-term stability are essential. Understanding why this is the case requires more than a slight overview of language features. It requires an examination of the historical context, underlying design principles, and practical consequences of using the language. This paper examines C++ from multiple perspectives. First, the historical development of the language, highlighting how practical systems programming challenges shaped its evolution. Particular attention is given to the role of Bjarne Stroustrup and his long-standing influence on both the language design and its standardization process (Stroustrup, 2013). Building on this foundation, the paper discusses the core design principles of C++, including its multi-paradigm nature and the principle of zero-overhead abstractions, which continue to guide its development (Stroustrup, 1995; Stroustrup, 2013). Beyond theory, the paper places strong emphasis on practical application. Common domains of use are examined to illustrate where C++ remains particularly well suited and where its complexity causes complications and costs. This discussion is complemented by a case study in the form of a command-line search tool implemented as part of the *Workshop: Programming Languages* task (2024). The implementation serves as a concrete example through which language features, tooling, and performance trade-offs can be discussed in a realistic setting. Rather than presenting C++ as a universally optimal solution, this paper adopts a balanced perspective. By combining historical analysis, design discussion, and hands-on implementation, it aims to show that the strengths and weaknesses of C++ are closely intertwined. The language's continued relevance lies not in simplicity or ease of use, but in its ability to make costs explicit and to give programmers precise control over performance and resources when such control is required.

## 2 Historical Development of C++

### 2.1 Origins and Early Motivations

The C++ programming language originated at the end of the 1970s as a response to practical problems encountered in systems programming. Stroustrup (1999) describes C++ as an extension of the C programming language, initially known as *"C with Classes"*, designed to provide data abstraction and stronger structuring mechanisms without sacrificing performance or compatibility with C. Early versions of C++ were documented before formal standardization in internal reference manuals. In the *C++ Programming Language*

— *Reference Manual*, Stroustrup characterizes C++ as "a general purpose programming language designed to make programming more enjoyable for the serious programmer" (Stroustrup, 1984). This formulation illustrates that C++ was designed from the outset as a general-purpose language rather than merely an incremental extension of C. The development of C++ was driven by real-world application needs rather than by purely theoretical language design considerations (Stroustrup, 2013). From its inception, the language aimed to support efficient low-level programming while enabling programmers to manage increasing software complexity (Stroustrup, 2013).

## 2.2 Role of Bjarne Stroustrup

The historical development of C++ is closely associated with the work of Bjarne Stroustrup. Who served both as the language's original designer and as a long-term contributor to its evolution. Stroustrup emphasizes his involvement in the language's design and standardization process, including participation in the ISO C++ standards committee (Stroustrup, 2013). Stroustrup consistently argued against defining C++ as a language restricted to a single programming paradigm. Instead, he positioned C++ as a multi-paradigm language capable of supporting procedural, object-oriented, and generic programming styles (Stroustrup, 1995).

## 2.3 Major Stages of Standardization

The growing popularity of C++ made formal standardization inevitable. According to Stroustrup (2013), the first ISO standard, C++98, provided a stable definition of both the core language and its standard library. Later standards followed an evolutionary approach. Brunner and Porkoláb (2017) characterize the evolution of C++ as a gradual increase in the abstraction level offered by the language, while preserving backward compatibility. A major milestone was C++11, which significantly expanded the language and library without fundamentally altering its original design goals (Stroustrup, 2013).

## 2.4 Evolution of Language Philosophy

Even though C++ has grown a lot, the underlying philosophy has remained. A central principle is that abstractions should not impose unnecessary runtime costs, summarized by Stroustrup as *"what you don't use, you don't pay for"* (Stroustrup, 2013). In later reflections, Stroustrup argues that many misconceptions about C++ stem from outdated views which do not consider decades of evolution. Modern C++ represents an enhanced realization of long-standing design ideals, offering improved safety and expressiveness without sacrificing efficiency (Stroustrup, 2025). In *21st Century C++*, Stroustrup (2025) identifies resource management and lifetime control as central goals of the language.

# 3 Core Concepts and Design Principles of C++

## 3.1 Type System and Abstraction Mechanisms

C++ is a statically typed language, meaning that the type of every entity must be known at compile time (Stroustrup, 2013). This enables a large compile-time checking and optimization. The type system provides equal support for built-in and user-defined types, enabling programmers to construct abstractions that integrate seamlessly with low-level operations (Stroustrup, 2025). Abstraction mechanisms such as classes and templates allow developers to express high-level concepts without hiding performance costs (Stroustrup, 2013).

## 3.2 Selected Central Features

Classes serve as the primary abstraction mechanism in C++, supporting encapsulation and deterministic resource management (Stroustrup, 2013). Templates enable generic programming and allow algorithms and data structures to be written independently of specific types while remaining efficient (Stroustrup, 1995). The C++ object and memory model ensures deterministic construction and destruction of objects. Stroustrup (2013) emphasizes that language rules governing object lifetime are essential for reliable resource management.

## 3.3 Distinctive Aspects Compared to C

Although C++ arised as an extension of C, Stroustrup explicitly rejects the notion of a combined "C/C++" language. He criticizes this misconception as hiding fundamental differences in design philosophy and expressive power (Stroustrup, 2025). Compared to C, C++ offers stronger abstraction mechanisms, a richer type system, and systematic resource management while retaining low-level control and efficiency (Stroustrup, 2013).

# 4 Applications and Use Cases

## 4.1 Common Domains of Use

Since its inception, C++ has been closely associated with application domains in which performance, resource control, and close interaction with hardware are central concerns. Unlike languages that target a narrowly defined programming paradigm or domain, C++ was intentionally designed to support a range of programming styles and application areas (Stroustrup, 1995). This design decision has had a lasting impact on the domains in which the language is commonly employed. Historically, C++ gained prominence in systems programming, including operating systems, compilers, networking software, and embedded systems (Stroustrup, 1999). Another major area of application is the development of libraries and infrastructure components. C++ is often used to implement foundational libraries that are later consumed by applications written in other languages. The combination of strong static typing, deterministic object lifetimes, and fine-grained control over memory makes C++ particularly suitable for this role (Stroustrup, 2013). In this context, the language's complexity is often accepted as a necessary cost, since library authors can hide complicated details behind stable interfaces, and make them easy to use. Game engines, real-time simulation, and high-performance computing represent further domains in which C++ continues to play a dominant role. These applications require predictable performance characteristics and direct control over system resources. While higher-level languages may offer faster development cycles, they often fail to meet the strict latency and throughput requirements imposed by such systems. As Stroustrup notes, C++ was never intended to maximize ease of use at all costs, but rather to enable efficient solutions where lower-level control remains essential (Stroustrup, 1999). A key factor in C++'s suitability for systems-level and performance-critical domains is the introduction of a formally specified memory and concurrency model in C++11. Prior to this standard, threads and memory ordering were largely implementation-dependent, limiting the reliability of concurrent systems code. Nienhuis, Memarian und Sewell (2016) show that the C++11 concurrency model "balances two goals: it is relaxed enough to be efficiently implementable ... and strong enough to give useful guarantees to programmers" (p. 1). Such guarantees are essential for domains involving parallel execution, low-level hardware interaction, and predictable real-time performance.At the same time, it is important to recognize the limits of C++'s applicability. In domains where rapid prototyping, dynamic adaptability, or minimal cognitive overhead are primary concerns, C++ is increasingly avoided. The language's expressive power comes at the price of complexity, which can slow development and increase maintenance costs if not managed carefully. This trade-off is not a flaw but a consequence of C++'s design goals, which prioritize performance and flexibility over simplicity (Stroustrup, 1995). Consequently, modern software systems often adopt a hybrid approach, using C++ for performance-critical components while relying on higher-level languages for orchestration and user-facing logic.

## 4.2 Possible Case Studies

A good illustration of C++'s strengths and limitations can be found in the implementation of the C++-Program as part of the Workshop (Workshop: Programming Languages, 2024). The program, conceptually similar to established tools such as grep, required recursive directory traversal, regular expression matching, and efficient handling of file input. These requirements align closely with the traditional application domains of C++,

particularly systems-level utilities.
The choice of C++ for this task highlights several practical advantages. The standard library provides facilities for filesystem traversal and regular expression processing that allow developers to express complex behavior without resorting to platform-specific code. Furthermore, deterministic resource management enables precise control over file handles and memory usage, an important consideration when processing large directory trees. These characteristics reflect Stroustrup's emphasis on providing abstractions that integrate seamlessly with low-level operations (Stroustrup, 2013). At the same time, the case study exposes the costs associated with using C++ for relatively small utility programs. Compared to scripting languages commonly used for similar tasks, the implementation effort is significantly higher. Decisions regarding data structures, error handling, and library usage require careful consideration, even when relying on modern C++ facilities. From a broader perspective, the search tool shows how C++ is used in todays software development. C++ performs best in Situations that require high and predictable performance, careful resource management, and integration with the operating system. However, its use must be justified against alternative solutions that may offer greater productivity at the cost of efficiency. As such, the case study shows the view that C++ is best understood not as a universal solution, but as a specialized tool whose strengths become apparent in carefully chosen application domains (Stroustrup, 1999).

# 5 Tooling and Ecosystem

## 5.1 Build Systems

The tooling ecosystem of C++ cannot be understood independently of the language's compilation model and design philosophy. C++ is a statically compiled language with a strong emphasis on efficiency and explicit control over translation units, which makes the build process an integral part of software development rather than a transparent background activity (Stroustrup, 2013). As a result, build systems are not just secondary tools but essential components of the C++ development workflow. Stroustrup emphasizes that C++ was designed to coexist with existing systems and toolchains, rather than to impose a uniform, centralized development environment (Stroustrup, 1999). This design choice has had long-term consequences for build tooling. Instead of a single standard build system, the C++ ecosystem developed a variety of tools that reflect different usage scenarios, platforms, and organizational constraints. The absence of a mandated build system is therefore not an oversight, but a direct consequence of the language's commitment to flexibility and broad applicability. The workshop task specified the development of an implementation as a standalone command-line tool, intended to run in a Unix environment (Workshop: Programming Languages, 2024). In this context, build tooling plays a practical role: transforming source code into an executable across different systems. Even for a relatively small program, the need to manage compiler invocation, standard conformance, and filesystem integration illustrates that build configuration is inseparable from the development process in C++. Conceptually, the complexity of C++ build systems mirrors the language's multi-paradigm nature. Since C++ supports procedural, object-oriented, and generic programming styles, build configurations need to handle diffrent compilation strategies and dependencies (Stroustrup, 1995). Consequently, the learning curve is often associated as a barrier to entry, yet it also reflects the language's capacity to scale from small utilities to large, performance-critical systems.

## 5.2 Package Managers

Closely related to build systems is the issue of dependency management. Unlike languages that introduced centralized package repositories and standardized distribution mechanisms, C++ historically relied on system-level libraries and manual integration of external code (Stroustrup, 2013). This reflects the language's origins in systems programming, where tight control over dependencies and binary interfaces was considered essential. Stroustrup argues that C++ was never intended to enforce a single programming style or workflow, but rather to support a wide range of use cases (Stroustrup, 1995). This design choice extends to package management. The diversity of application domains and deployment environments makes it difficult to define a one-size-fits-all solution for dependency handling. As a result, the C++ ecosystem exhibits fragmentation in this area, with multiple competing approaches rather than a unified standard. In the context of the workshop task, the implementation intentionally avoided external dependencies and relied exclusively on the C++ standard library (Workshop: Programming Languages, 2024). This decision simplified dependency management and ensured that the program could be built and executed without additional configuration. It also makes visible a common tension in the ecosystem: while minimal dependencies reduce complexity for small projects, larger systems often require extensive third-party libraries, reintroducing the challenges that package managers aim to address. The lack of a universally adopted package management solution can be interpreted not as a failure of the ecosystem, but as a reflection of C++'s foundational principles. The language prioritizes control and flexibility over convenience, accepting increased complexity as a trade-off for broader applicability (Stroustrup, 2013). This helps explain why dependency management remains a recurring topic of debate within the C++ community.

## 5.3 Relevant Tooling

Beyond build systems and package managers, the broader tooling ecosystem encompasses compilers, debuggers, and analysis tools that support the development of reliable and efficient software. In C++, these tools play a particularly prominent role due to the language's emphasis on static typing, explicit resource management, and performance optimization (Stroustrup, 2013). Stroustrup notes that C++ was designed to perserve low-level control and manual optimization when necessary, rather than abstracting such concerns away entirely (Stroustrup, 1995). This design philosophy implies that tooling must expose detailed information about program behavior, memory usage, and execution flow. As a consequence, effective C++ development often depends heavily on external tools to diagnose errors, analyze performance bottlenecks, and guarantee correctness. The implementation developed for the workshop illustrates this dependency in a practical setting. Even though the program itself is relatively compact, reasoning about file traversal, regular expression matching, and memory usage requires careful inspection and testing. The task specification encouraged reflection on performance and optimization considerations, reinforcing the importance of tooling as part of the development process rather than as an afterthought (Workshop: Programming Languages, 2024). Historically, the evolution of C++ tooling has followed the language's gradual expansion. As the language incorporated more abstraction mechanisms and library facilities, the surrounding tools evolved to support these features without compromising performance (Stroustrup, 1999). This co-evolution underscores a central insight: in C++, the language and its tooling form an inseparable ecosystem, shaped by the same design goals and constraints.

## 6 Implementation of the Search Tool

### 6.1 Overview of Task and Goals

The implementation task required the development of a command-line search tool that recursively scans files and directories for a given regular expression pattern and outputs matching lines together with optional contextual information. According to the task specification provided in the *Workshop: Programming Languages* (2024), the program should behave similarly to established tools such as `grep`, while not necessarily replicating all of their features. The core functional requirements included recursive directory traversal, regular expression matching, exclusion of binary files, optional context lines before and after matches, and configurable output formatting. In addition to the implementation itself, the accompanying paper is explicitly required to include a detailed account of the development process, design decisions, and optimization considerations. The implementation and its analysis, emphasizing reflection on language features, libraries used, and performance considerations (*Workshop: Programming Languages*, 2024). Consequently, the implementation of the search tool is not treated as a purely technical exercise, but as a case study demonstrating practical use of the programming language. The primary goal of the implementation was therefore twofold: first, to satisfy the functional requirements of the search tool as specified in the task description; and second, to demonstrate idiomatic use of modern C++ facilities, particularly those provided by the standard library, in accordance with the design principles discussed by Stroustrup (2013, 2025).

## 6.2 Key Design Choices

A central design decision was to rely exclusively on facilities provided by the C++ standard library rather than external dependencies. This aligns with Stroustrup's emphasis on the standard library as an integral part of the language, designed to support common programming tasks efficiently and safely (Stroustrup, 2013). The implementation uses `std::filesystem` for directory traversal, `std::regex` for pattern matching, and standard container and algorithm facilities such as `std::vector`, `std::map`, and `std::sort`. Another key decision concerns the overall program structure. The tool is implemented as a single executable that parses command-line arguments, traverses the file system, processes files line by line, and produces formatted output. This structure mirrors the task specification, which requires a command-line interface capable of handling multiple options and arguments (*Workshop: Programming Languages*, 2024). The design deliberately avoids complex class hierarchies in favor of straightforward procedural organization, reflecting the multi-paradigm nature of C++ as described by Stroustrup (2013). The decision to read entire files into memory before processing them was made to simplify context handling. By storing each file's contents as a vector of strings, the implementation can easily determine which lines precede or follow a match, as required by the `-A`, `-B`, and `-C` options specified in the task description (*Workshop: Programming Languages*, 2024). This choice prioritizes clarity and correctness over minimal memory usage, which is acceptable given the educational context of the task. This design choice has direct implications for memory usage. Reading complete files into memory scales with the total size of the processed input and is therefore not optimal for very large files. However, this approach significantly simplifies the implementation of context handling and the merging of overlapping match regions. Within the educational scope of the assignment, this trade-off was considered acceptable, as it prioritizes clarity and correctness over memory efficiency.

## 6.3 Use of C++-Specific Features

The implementation makes extensive use of C++-specific features that support safety, expressiveness, and efficiency. File handling is performed using `std::ifstream`, which exemplifies Resource Acquisition Is Initialization (RAII): file resources are acquired during object construction and released automatically when the stream object goes out of scope. Stroustrup highlights RAII as a fundamental idiom in C++, enabling reliable resource management without explicit cleanup code (Stroustrup, 2013). Directory traversal is implemented using `std::filesystem::recursive_directory_iterator`, introduced to provide a portable and type-safe interface for file system operations. In *21st Century C++*, Stroustrup emphasizes the importance of modern standard library facilities that replace platform-specific or error-prone low-level code with well-defined abstractions (Stroustrup, 2025). The use of `std::filesystem` directly reflects this design philosophy. Regular expression matching is handled using `std::regex` with ECMAScript syntax. The implementation conditionally applies the `std::regex::icase` flag when the `-i` option is specified, thereby fulfilling the requirement for case-insensitive search as described in the task specification (*Workshop: Programming Languages*, 2024). This demonstrates how C++ integrates powerful text-processing facilities into the standard library. While `std::regex` provides a portable and expressive solution that integrates seamlessly with the standard library, it is known to introduce non-trivial runtime overhead and implementation-dependent performance characteristics. More specialized search algorithms or optimized pattern matching libraries could achieve significantly higher throughput. The use of `std::regex` therefore reflects a deliberate focus on standard compliance and readability rather than maximal performance. The use of standard containers such as `std::vector` and `std::map` illustrates the principle of equal support for built-in and user-defined types. Stroustrup stresses that C++ abstractions should be efficient enough to be used pervasively, rather than reserved for special cases (Stroustrup, 2013). An ordered associative container is used to store file contents in a deterministic order. Although this choice introduces additional overhead compared to unordered alternatives, it ensures reproducible output and predictable iteration order. This property was considered beneficial for a reference-style implementation intended for analysis and comparison.

## 6.4 Handling Recursion, Traversal, and Matching

Recursive traversal of directories is a core requirement of the task.
The use of `std::filesystem::recursive_directory_iterator` allows the program to visit all regular files under the specified root path, including nested subdirectories, in a manner consistent with the task specification (*Workshop: Programming Languages*, 2024). Non-regular files are explicitly skipped to avoid undefined behavior. Binary files are excluded from the search by performing a heuristic check for null bytes in the initial portion of each file. While the task specification only requires that binary files be excluded, it does not mandate a specific detection method (*Workshop: Programming Languages*, 2024). The chosen approach balances simplicity and effectiveness. It should be noted that this method is heuristic in nature and may lead to false positives or negatives in edge cases. The simplification is nonetheless sufficient to meet the requirements of the task and avoids the complexity of full encoding or file-type analysis. Pattern matching is performed line by line using `std::regex_search`. Matching line indices are stored and subsequently

used to compute context ranges based on the user-specified options. Overlapping context ranges are merged to prevent duplicate output, a behavior explicitly described in the example outputs of the task specification (*Workshop: Programming Languages*, 2024). Output formatting follows the conventions outlined in the task description, including the use of separators to determine matching lines from context lines and the insertion of delimiters between non-adjacent context blocks (*Workshop: Programming Languages*, 2024). The optional `-no-heading` mode alters the output format to include the filename on each line, demonstrating conditional behavior driven directly by command-line options.

### 6.5 Reflection on Task Compliance

Overall, the implementation satisfies the core functional requirements defined in the programming task. Recursive search, regular expression matching, context handling, and binary file exclusion are all implemented in accordance with the specification (*Workshop: Programming Languages*, 2024). While certain optional features, such as colored output, are not implemented, the task description explicitly allows some flexibility in feature completeness. The implementation processes files sequentially and does not employ parallelization or streaming-based processing. While such techniques could improve performance on large directory trees, they were intentionally excluded to keep the implementation comprehensible and focused on functional correctness rather than maximal throughput. From a language perspective, the implementation illustrates Stroustrup's assertion that the value of a programming language lies in the quality of its applications (Stroustrup, 2025). By using modern C++ facilities and standard library components, the program achieves a balance between clarity, safety, and expressiveness, reflecting the design goals articulated in *The C++ Programming Language* (Stroustrup, 2013). These design decisions directly influence the performance characteristics of the implementation, which are examined in the following section.

# 7 Performance Observations

### 7.1 Basic Profiling

Performance observations were conducted at a conceptual and structural level rather than through precise runtime measurements. This approach is consistent with Stroustrup's view that performance considerations should be grounded in an understanding of program structure, cost models, and algorithmic behavior before attempting low-level optimization (Stroustrup, 2013). Rather than reporting raw timing results, the analysis focuses on identifying performance-relevant design decisions and their implications. The implemented tool follows a clear processing pipeline consisting of recursive directory traversal, file filtering, line-based input processing, regular expression matching, and formatted output generation. Examining this pipeline makes it possible to reason about dominant cost factors without introducing artificial measurement data that would provide limited insight in an educational setting.

## 7.2 Main Bottlenecks and Optimization Considerations

In practice, the traversal cost increases with the size of the directory tree, not with the number of matches, and therefore becomes the main cost for large inputs. File handling constitutes another significant performance consideration. Each non-binary file is opened using `std::ifstream` and read completely into a `std::vector<std::string>` before pattern matching is performed. As discussed in the previous chapter, this design simplifies context handling but increases memory usage and introduces additional copying overhead. Regular expression matching using `std::regex_search` represents another major cost factor. While `std::regex` offers portability and expressiveness, it is not optimized for maximum throughput. This choice reflects a deliberate emphasis on correctness and standard compliance over specialized performance optimization, in line with the educational goals of the assignment. The merging of overlapping context ranges introduces additional computation but remains negligible compared to file I/O and pattern matching. This processing step improves output readability and directly mirrors the example outputs described in the task specification (*Workshop: Programming Languages*, 2024). Its computational complexity is linear in the number of matches and does not constitute a practical performance bottleneck.

## 7.3 Reference to Established Tools and Trade-offs

The task specification explicitly references established tools such as `grep` and `ripgrep` as potential points of comparison (*Workshop: Programming Languages*, 2024). These tools employ advanced optimization techniques, including memory-mapped I/O and highly specialized pattern-matching engines, and are therefore capable of significantly higher performance than the presented implementation. From a language-design perspective, Stroustrup argues that the value of a programming language should not be judged by isolated micro-benchmarks, but by the quality and maintainability of the applications developed with it. As he states: "The value of a programming language is in the quality of its applications" (Stroustrup, 2025). Viewed in this light, the implemented search tool should not be evaluated as a competitor to industrial-strength utilities, but as a demonstration of idiomatic C++ usage under clearly defined constraints. The implementation illustrates how modern C++ facilities—such as RAII-based resource management, standard containers, and filesystem abstractions—can be combined to produce a correct, maintainable, and comprehensible program that fulfills the specified requirements. Stroustrup further cautions against premature optimization, emphasizing that performance improvements should be guided by measurement and understanding rather than speculation (Stroustrup, 2013). Given the educational scope of the task and the explicit allowance for partial feature completeness, the absence of aggressive low-level optimizations is consistent with both the task requirements and the underlying design philosophy of C++.

### 7.4 Summary of Performance Findings

In summary, the performance characteristics of the implemented search tool are dominated by file system traversal, file I/O, and regular expression matching—factors that are inherent to the problem definition. The implementation adheres closely to the task specification and demonstrates a clear, idiomatic use of C++ standard library facilities. While more advanced optimizations would be possible, particularly when compared to specialized tools, the current design represents a balanced solution that prioritizes correctness, readability, and maintainability. This balance is consistent with both the educational goals of the task (*Workshop: Programming Languages*, 2024) and Stroustrup's design principles emphasizing explicit trade-offs and disciplined performance reasoning (Stroustrup, 2013; Stroustrup, 2025).

## 8 Conclusion

This seminar paper set out to examine C++ not only as a programming language, but as the result of a long sequence of design decisions shaped by practical constraints. The analysis showed that C++ did not emerge from purely theoretical considerations, but from concrete needs in systems programming. Many of its defining characteristics, such as its emphasis on efficiency, compatibility, and explicit control, can be traced directly back to these origins (Stroustrup, 2013). The discussion of design principles and core language concepts highlighted that C++ deliberately resists being reduced to a single programming paradigm. Its support for procedural, object-oriented, and generic programming reflects a broader philosophy that prioritizes flexibility over uniformity (Stroustrup, 1995). This flexibility, however, comes at the cost of increased complexity, a trade-off that becomes particularly visible in modern software development contexts. The implementation of a command-line search tool demonstrated how modern C++ facilities can be combined to produce a correct and maintainable program using only the standard library. At the same time, the analysis of tooling and performance made clear that such implementations inevitably involve trade-offs. Decisions that improve clarity and correctness—such as relying on standard abstractions or avoiding aggressive optimization—also influence performance characteristics. These effects are not accidental, but a direct consequence of the language's design philosophy, which encourages programmers to reason explicitly about costs rather than hiding them (Stroustrup, 2013; Stroustrup, 2025). In summary, C++ emerges neither as an outdated language nor as a universally appropriate solution. Its strengths become most apparent in carefully chosen domains where performance, predictability, and control justify its complexity. The case study presented in this paper illustrates that, when used deliberately and with an understanding of its trade-offs, C++ remains a powerful and relevant tool. Its continued evolution suggests that its role in software development will persist, not because it is simple, but because it enables disciplined and explicit reasoning about software systems.

## 9 References

Stroustrup, B. (1984). *The C++ programming language — reference manual.* AT&T Bell Laboratories.

Stroustrup, B. (1995). *Why C++ is not just an object-oriented programming language.* OOPSLA Addendum to the Proceedings of OOPSLA '95.

Stroustrup, B. (1999). *An overview of the C++ programming language.* In *Handbook of Programming Languages* (Vol. 1). Macmillan Technical Publishing.

Brunner, T., & Porkoláb, Z. (2017). *Programming language history: Experiences based on the evolution of C++.* In *Proceedings of the 10th International Conference on Applied Informatics* (pp. 63–71).

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Addison-Wesley.

Nienhuis, K., Memarian, K., Sewell, P. (2016). An operational semantics for C/C++11 concurrency. In Proceedings of OOPSLA 2016 (pp. 111–131). ACM.

Stroustrup, B. (2025). *21st century C++.* Columbia University. Technical Report P3650R0.

Workshop: Programming Languages. (2024). *Programming task and paper assignment.* Course material.