Automated Migration of Legacy Code from the C++14 to C++23 Standard

Aleksander Świniarski[a] and Anna Derezińska[b]
Warsaw University of 5/19, Warsaw, Poland
{aleksander.swinarski.stud, anna.derezinska}@pw.edu.pl

Keywords: Transpiler, Source-to-Source Compiler, C++, Legacy Code, C++14, C++23.

Abstract: The continuous development of the C++ programming language results in changes in many programming features from one version to another. Therefore, we face a growing increase in maintenance and evolution costs. To address this problem, a set of removed and deprecated programming features was examined, and automating of the feature migration was proposed. A transpiler has been developed that transforms a C++ code from a legacy form to its latest standard. The CppUp tool translates a C++14 program into its equivalent C++23. The current version of the tool supports 17 removed and 3 deprecated features. The restrictions of the tool limit its practical application, but the experiments conducted on seven real-world programs confirmed the reliability and usability of the transpiler.

1 INTRODUCTION

Legacy software systems could be very important in the operational strategy of business processes and industrial practice. Maintenance of such systems and manual migration between different dialects of a programming language are a time-consuming and costly activity (Sneed and Verhoef, 2020).

The C++ programming language, originated in the late 1970s, is still widely used for software development (ISOCPP, 2024). It is especially beneficial when we challenge requirements of high performance and low energy consumption. Since the revolutionary change in 2011, every several years new versions of the language with a set of feature improvements have been announced (Bancila, 2024).

In this paper, we address the problem of a C++ program that migrates from a legacy form to the latest one. C++14 was chosen as the starting point for migration due to its widespread use in real-world projects, as highlighted in the 2024 C++ Developer Survey (ISOCPP, 2024), which shows a significant number of developers still rely on this version. All modifications between consecutive variants from C++14 (ISO/IEC, 2014) to C++23 (ISO/IEC, 2024) have been revised. Current research has focused on those that hinder program development the most and fall into the removed and deprecated categories. For the selected features, source-to-source transformation guidelines have been developed.

To support automated code migration, a CppUp transpiler has been designed and implemented (Cooper, 2011). The tool transforms a program from one dialect to another while maintaining its functionality. Its application reduces the migration effort and minimizes the number of errors that could be introduced during this process.

CppUp has been evaluated on a set of programs. Unit tests and tests using real-world programs dealt with various coding practices corresponding to the migrated features. The evaluation of the transpiler confirmed the reliability of the transformation within the limitations of the current solution.

The CppUp code, together with its unit tests and the applications used in the tool evaluation, is available at (Świniarski, 2024).

The main contributions of the paper are:

- Examining examples of legacy programming features and a way of their migration;
- Development of the CppUp tool that supports 17 removed and 3 deprecated features in the transformation from C++14 to C++23;
- Experimental evaluation of the transpiler.

[a] https://orcid.org/0009-0008-4564-9061
[b] https://orcid.org/0000-0001-8792-203X

Świniarski, A. and Derezińska, A.
Automated Migration of Legacy Code from the C++14 to C++23 Standard.
DOI: 10.5220/0013298000003928
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 549-556
ISBN: 978-989-758-742-9; ISSN: 2184-4895
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

The paper is organized as follows. In the next section, we briefly describe basic problems of code transpilers and review related work. Section 3 contains an explanation of the selected programming features transformed between different versions of the C++ standards. The CppUp tool is presented in Section 4. In Section 5, we discuss the evaluation of the tool. Finally, Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

Transpilers, or source-to-source compilers, automate code migration by translating code between dialects or languages while preserving functionality. They ensure that the input and output code remain at the same abstraction level.

2.1 Transpilers in Research and Industry

Many transpilers have been developed for different purposes. A systematic review of transpilers and their application can be found in (Bastidas Fuertes, Pérez, and Meza Hormaza, 2023). It has been reported that in the industry, transpilers are primarily used for (i) code reusing and migration strategy for legacy platforms, (ii) achieving compatibility with end-user and mobile platforms, and (iii) generating language extensions, mainly as a superset of Javascript.

Problems of legacy code translation were faced in (Schnappinger and Streit, 2021). A system written in Natural, Cobol, and Assembler was migrated to its corresponding system in Cobol on Linux and Java. The legacy code was parsed using a grammar defined with ANTLR (ANTLR, 2024). A custom transpilation was promoted, that is, only basic simple transformation rules were implemented, while the legacy code could have been inspected in transpilation time, and complicated cases resolved by a user. Consequently, new rules were added to the grammar and translation.

Transpilers can operate within the same language family. For example, a program in the strongly typed TypeScript language needs to be compiled with the tsc transpiler in JavaScript form, which can be executed (TypeScript, 2024).

Others deal with languages with similar functionality running in different environments, such as Kotlin (Android) and Swift (iOS) supported by unidirectional and bidirectional transpilers (Schneider and Schultes, 2022).

An example of a multi-platform approach is Haxe, an object-oriented and strongly typed programming language, which framework is associated with transpilation facilities to C#, Java, C++, Python, and PHP (Haxe, 2024).

While general functionality should be preserved, source-to-source transformation could be associated with enhancement of different non-functional requirements, namely: supporting parallel execution, reducing power consumption, avoiding selected programming concepts, etc.

The transformation of C programs into Rust is supported by different tools, including the C2Rust transpiler. However, these tools preserve the unsafe semantics of C. The authors of CRustS, described in (Ling, et al., 2022), focus on the safety issues available in Rust. The approach is based on a set of source-to-source transformation rules, both preserving strict semantics (198 rules) and approximating semantics with a more safe code of Rust (22 rules).

2.2 Migration of C++ Programs

Several tools have been developed to facilitate code migration of C++ programs.

Originally, C++ programs were translated into C using the Cfront cross-translator. Hence, existing C compilers could be utilized to develop the final code.

Code migration could be performed in different directions, that is, ‘from’ or ‘to’ legacy versions. The latter case is described in (Antal, et al., 2016), where developers wanted to use new features of C++11, at that time, while the code was supposed to be compatible with C++03 used by an industrial partner. Using (LLVM Clang, 2024), a tool was developed that backported a large subset of C++11 features to its older legacy version.

Parallel processing was also supported by C++ transpilers. A Togpu tool was developed to transform C++11 into parallel CUDA to lower the entrance barrier to GPU (Marangoni and Wischgoll, 2016). The OP2 framework enables translation of C/C++/Fortran programs into different parallel models, e.g. CUDA, MPI (Chen, et al., 2024).

In high-level synthesis, C/C++ programs are converted into appropriate domain languages, such as Verilog (Xu et al., 2024). These kinds of program do not cope with new programming features, as the number of allowed programming structures is usually strongly restricted.

To the best of our knowledge, none of the transpilers supports a C++ program migration in the scope addressed in this paper.

3 MIGRATION FROM C++14 TO C++23

The C++ language has evolved significantly beyond C++14, with C++17 (ISO/IEC, 2017), C++20 (ISO/IEC, 2020), and C++23 offering improved efficiency, readability, safety, and performance. Yet, many codebases still rely on older standards, hindering maintainability and compatibility. Upgrading these legacy projects is essential to optimize applications and ensure sustainability.

3.1 Taxonomy of Migration Features

When migrating a C++ codebase to a newer standard, it is crucial to understand the added, modified, deprecated, and removed features. These can be grouped into the following *feature categories:*

- *Removed* - Elements that have been entirely removed from the language. Code utilizing these features will result in compilation errors;
- *Deprecated* - Features that are marked for potential removal in future standards. While still supported, their use generates warnings during compilation;
- *New* - Additions to the language that introduce new capabilities, improve performance, or enhance code expressiveness;
- *Modified* - Existing features that have undergone changes in syntax or behavior. Code using these features may require adjustments to align with the updated definitions to maintain compatibility;
- *Miscellaneous* - Category that encompasses various minor additions, enhancements, or changes that do not fit into the major categories, such as core language features, standard library updates, or syntax changes.

While migrating from C++14 to C++23, *removed* and *deprecated* features pose the most immediate concerns: removed features cause compilation errors, while deprecated ones generate warnings that, if ignored, can accumulate technical debt.

Table 1 summarizes the C++ features deprecated or removed between C++17 and C++23, showing the standard version for each change. It is based on Annex C of the C++ standards (C++14–C++23) and additional documents from the ISO C++ committee, including (Köppe, 2018) and (Köppe, 2020).

Some features underwent a two-step process: first deprecated (D) in one standard version, and then removed in a later one. As a result, they appear twice in the table, reflecting each stage in their timeline. Counting each row separately yields 42 changes, but consolidating duplicates reduces it to 35 unique changes.

In the current prototype, only a selected subset of these features is implemented, with the rest deferred due to complexity. Table 1 labels implemented features as “Yes” and unimplemented ones as “No”.

3.2 Overview of Transformations

As an example, the following sections present features 5. and 9. for migrating the C++14 code to the C++23 standard. For each feature, the rationale behind its removal or deprecation and the method to replace it with a C++23-compatible equivalent are discussed. The approach is supported by insights from documentation and related discussions.

3.2.1 Smart Pointer std::auto_ptr (Id. 5.)

The `std::auto_ptr` is a smart pointer that manages dynamically allocated objects, automatically deleting them when destroyed. It grants unique ownership of the object it points to. However, `std::auto_ptr` has problematic copying semantics: when copied, ownership transfers to the destination pointer, leaving the source as `nullptr`. This violates conventional copy semantics, where copies are expected to be equal and independent, leading to potential unexpected behavior (Lavavej, 2014).

Due to these issues, `std::auto_ptr` was deprecated in C++11 and removed in C++17, replaced by `std::unique_ptr`, which uses move semantics to ensure a consistent and safe ownership transfer. Migrating involves replacing `std::auto_ptr` declarations with `std::unique_ptr` and updating any copy operations to use `std::move`.

For example:

```
   // Original code
   std::auto_ptr<int> ptr1(new int{1});
   std::auto_ptr<int> ptr2(ptr1);
   // Updated code
   std::unique_ptr<int> ptr1(new
int{1});
   std::unique_ptr<int>
ptr2(std::move(ptr1));
```

3.2.2 Binary Function Binders (Id. 9.)

In earlier versions of C++, `std::bind1st` and `std::bind2nd` (from `<functional>`) created unary function objects by binding one argument of a binary function.

However, these binders had limitations and were deprecated in C++11, then removed in C++17. They have been replaced by the more flexible `std::bind` and lambda expressions, which offer a clearer syntax (Lavavej, 2014).

Migrating from `std::bind1st` or `std::bind2nd` to `std::bind` involves updating the argument list to specify which argument is bound and which remains dynamic using `std::placeholders::_1`.

- For `std::bind1st`, bind the first argument by placing the constant value as the second argument and `std::placeholders::_1` as the third argument;
- For `std::bind2nd`, bind the second argument by placing `std::placeholders::_1` as the second argument, and the constant value as the

Table 1: List of features that were removed or deprecated throughout C++14 to C++23.

Id	Feature name	C++ standard	Feature category	Handled in CppUp
1.	Trigraphs	C++17	Removed	Yes
2.	Register keyword	C++17	Removed	Yes
3.	++ for Booleans	C++17	Removed	Yes
4.	throw(A,B,C)	C++17	Removed	Yes
5.	auto_ptr	C++17	Removed	Yes
6.	random_shuffle	C++17	Removed	Yes
7.	Function objects	C++17	Removed	Yes
8.	Function objects Wrappers	C++17	Removed	Yes
9.	Binary Function Binders	C++17	Removed	Yes
10.	Iostream Aliases	C++17	Removed	Yes
11.	Allocator Support From Function	C++17	Removed	Not
12.	Redeclaration of static constexpr Class Members	C++17	Deprecated	Not
13.	C Library Headers	C++17	Deprecated	Yes
13.	Ineffective “C++ versions” of compatibility headers	C++17, C++20	D, Removed	Yes
14.	Allocator<void>, Redundant Members of std::allocator	C++17	Deprecated	Not
15.	raw_storage_iterator	C++17	Deprecated	Not
15.	raw_storage_iterator	C++17, C++20	D, Removed	Not
16.	get_temporary_buffer	C++17	Deprecated	Not
16.	Temporary buffer API	C++17, C++20	D, Removed	Not
17.	is_literal_type	C++17	Deprecated	Yes
17.	is_literal_type	C++17, C++20	D, Removed	Yes
18.	std::iterator	C++17	Deprecated	Not
19.	<codecvt>	C++17	Deprecated	Not
20.	memory_order_consume	C++17	Deprecated	Not
21.	shared_ptr::unique	C++17	Deprecated	Yes
21.	shared_ptr::unique	C++17, C++20	D, Removed	Yes
22.	result_of	C++17	Deprecated	Yes
22.	result_of	C++17, C++20	D, Removed	Yes
23.	uncuaght_exception	C++17	Deprecated	Yes
23.	uncaught_exception	C++17, C++20	D, Removed	Yes
24.	noexcept-specifier throw()	C++20	Removed	Yes
25.	Functional adaptors and argument type class members	C++20	Removed	Yes
26.	Redundant members of allocator	C++20	Removed	Not
27.	Only's two complement representation for signed integers	C++20	Deprecated	Not
28.	Notion of POD type	C++20	Deprecated	Not
29.	Implicit lambda capture of this via [=]	C++20	Deprecated	Yes
30.	Comma operator in subscripting expressions	C++20	Deprecated	Yes
31.	Deprecate certain volatile qualifications	C++20	Deprecated	Not
32.	Shrinking basic_string::reserve	C++20	Deprecated	Yes
33.	Garbage collection support	C++23	Removed	Not
34.	aligned_union and aligned_storage	C++23	Deprecated	Not
35.	float_denorm_style, has_denorm_loss and has_denorm	C++23	Deprecated	Not

third.

An exemplary migration looks as follows:

```
   //Original code
   auto funFirst =
std::bind1st(std::multiplies<double>(),
pi/180.0);
   auto funSecond =
std::bind2nd(std::multiplies<double>(),
pi/180.0);
   //Updated code
   auto funFirst =
std::bind(std::multiplies<double>(),
pi/180.0, std::placeholders::_1);
   auto funSecond =
std::bind(std::multiplies<double>(),
std::placeholders::_1, pi/180.0);
```

4 CPPUP TRANSPILER

The CppUp transpiler is a tool designed to transform the code compatible with the C++14 standard into one that is compatible with the C++23 standard. The core aim of CppUp is to simplify code migration for developers who want to use the features provided in the newer standard without the need for manual code refactoring.

4.1 Requirements

The following section details the functional requirements (A–G) and non-functional requirements (H–M) for the CppUp transpiler.

A. *Parsing and Syntax Analysis* – Accurately parse valid C++14 syntax and provide clear error messages for syntax issues;
B. *Transformation Guidelines* – Implement robust methods to convert C++14 to C++23, focusing on 20 key features listed in Table 1, replacing deprecated elements with suitable alternatives;
C. *Preservation of Functionality* – Ensure that transformed code retains original functionality, readability, and maintainability;
D. *Standards Compatibility and Compliance* – Guarantee that the generated code adheres fully to the C++23 standard while maintaining backward compatibility with the g++14 compiler from GCC (GCC, 2024);
E. *Extensibility and Configurability* – Design a project so that the code transformations are easy to customize and add;
F. *Testing* – Include comprehensive testing to validate functionality and prevent regressions;
G. *User Interface and Experience* – Provide a user-friendly CLI with clear options, including verbose mode for detailed logs;
H. *Performance* – Ensure efficient transformation, handling large codebases quickly;
I. *Usability* – Offer an intuitive CLI and a detailed user manual;
J. *Reliability* – Deliver robust and accurate transformations that maintain code behavior, handle edge cases, and provide stability on UNIX systems;
K. *Maintainability* – Ensure modular, well-structured implementation for easy updates and debugging;
L. *Scalability* – Efficiently handle large projects with many files and complex structures;
M. *Interoperability* – Integrate seamlessly with build systems and development environments.

4.2 Design Overview

This section outlines the overall design of the CppUp transpiler, highlighting the key components and methodologies that enable its functionality.

4.2.1 Tools Used in Implementation

The CppUp transpiler was developed in C++23, leveraging modern language features for efficiency and maintainability. It uses the g++ compiler version 14 from the GNU Compiler Collection, which provides near-complete support for C++23.

ANTLR 4.13.1 was chosen for parsing and syntax analysis due to its flexibility, ease of use, and prior familiarity, which allows rapid prototyping. The build process is managed with CMake, simplifying configuration and cross-platform compilation.

ClangFormat from LLVM (LLVM Clang, 2024). Ensures that the generated code adheres to industry-standard styles, enhancing readability and consistency. Unit testing is performed with GoogleTest, providing a robust framework to maintain functionality as the project evolves.

4.2.2 Workflow of the Program

The operation of CppUp is divided into three main steps.

At first, the program reads command-line arguments to configure its behavior, such as specifying input and output paths or enabling optional features.

The second step of CppUp is to translate a C++14 code into its C++23 equivalent. It first creates an output directory for the resulting files and directories, then iterates through entities in the input path, performing operations based on each entity type:

- *Directory* – The program creates a corresponding directory with the same name in the output directory;
- *C++ File* – CppUp creates a file with the same name in the output directory, analyzes it to extract preprocessing directives and translation units, applies all implemented transformations, writes the updated code, and formats it using clang-format;
- *Non C++ File* – The transpiler by default copies the file with its name and contents to the directory for the resulting files.

In the last step, if specified, CppUp compiles all transpiled C++ files using g++14. Users can also request a complete build if the output forms a valid C++ program.

4.2.3 General Architecture of CppUp

The architecture of CppUp is built for modularity and extensibility, with each component serving a distinct purpose. The core class, CppUp, orchestrates the transformation process, manages file operations, generates transpiled code, and optionally compiles or builds the resulting project.

Parsing and syntax analysis are handled by CPP14Lexer and CPP14Parser, constructed from the C++14 grammar in ANTLR's grammar-v4. The lexer tokenizes the source code into elements like keywords, literals, and operators, while the parser generates a parse tree representing the code's syntactic structure.

The CppUpVisitor module handles code transformation, traversing the parse tree to identify and modernize constructs incompatible with C++23. It ensures that the transformed code is functionally equivalent while adhering to modern standards.

5 CPPUP EVALUATION

This section presents an evaluation of the CppUp transpiler. The evaluation encompasses both the controlled testing of individual modules and practical experiments conducted on real-world applications.

5.1 Testing of the Approach

To ensure CppUp's reliability, a testing strategy combined unit tests and real-world assessments. Unit tests validated individual modules, including file handling, code generation, syntax error management, and transformations for all 20 implemented features marked as “Yes” in the “Handled” column (Table 1).

5.2 Experiments on Real-World Applications

To assess the effectiveness of CppUp in practical scenarios, a diverse set of open-source C++14 projects was selected based on criteria such as language compliance, feature usage, codebase size, and available functionality verification methods. The projects ranged from small applications to large systems, thoroughly evaluating the tool's capabilities.

The testing methodology included the following:

1. *Baseline Compilation and Execution* – The projects were compiled and executed using their existing build systems to establish a functional baseline;
2. *Selective Transpilation* – Only files using the supported C++14 features were transpiled and reintegrated due to current limitations;
3. *Compilation with the C++23 Standard* – Modified projects were recompiled with GCC14 using the C++23 standard;
4. *Comparison of Results* – The outputs from the unit tests and the example applications were compared against the baseline to ensure functional consistency.

Table 2 summarizes the selected projects, while Table 3 details the experimental results, including metrics like total LOC, transpiled LOC, and the tested features. In particular, some features handled by CppUp were not tested, as no real-world projects that used all features were identified during the study. The projects in Table 3 are identified by their identifiers from Table 2.

5.3 Discussion of Results

The evaluation confirmed that CppUp effectively transpiles key C++14 features to C++23, maintaining functionality across diverse projects. Unit tests validated individual modules, while real-world tests demonstrated seamless integration and consistent results. However, challenges were noted. Selective transpilation was necessary due to the separate handling of preprocessing directives and translation units, limiting the number of files processed (Table 3). Additionally, the reduction in output lines of code (LOC) was due to clang-format enforcing consistent formatting rather than simplification of constructs.

Table 2: Real-World Projects Used in Experiments.

Id	Project name and origin	Description
I	CellularForms. Fogelman, M., https://github.com/fogleman/CellularForms	Implementation of Andy Lormans' conference paper on cellular growth (Lomas, 2014)
II	Flowcpp. Vanatasin, K., https://github.com/kittinunf/flowcpp	A header only implementation of the JavaScript Redux (Abramov, 2024)
III	Butterworth Filter Design. Ruotsi, R., https://github.com/ruohoruotsi/Butterworth-Filter-Design	Project that provides a collection of classes for designing high-order Butterworth filters
IV	Curvature filter. Gong, Y., https://github.com/YuanhaoGong/CurvatureFilter	Collection of algorithms developed by Yuanhao Gong during his PhD work (Gong, 2017)
V	CoRM. Taranov, K., https://github.com/spcl/CoRM.	Remote memory system designed to support data compaction over RDMA, developed as a part of research project presented at SIGMOD 2021 (Taranov, 2021)
VI	Microvolt. Deynega, A., https://github.com/deinega/microvolt	Program for modelling semiconductor devices
VII	Evhttpclient. Potter, J., https://github.com/jspotter/evhttpclient	HTTP client written in C++

Table 3: Results of experiments on real-world projects.

Project id	Project LOC (files)	Transpiled LOC (files) In	Transpiled LOC Out	Tests	Feature tested
I	2578 (41)	71 (1)	65	Procedural 3D Object Generator	22., 30.
II	450 (11)	450 (11)	474	Example of library usage	29., 30.
III	7307 (6)	832 (1)	725	Unit tests with 113 assertions	5., 23., 30.
IV	2224 (4)	2067 (2)	1998	Program for filtering exemplary images	2., 30.
V	8101 (37)	161 (2)	126	Binaries for managing remote memory systems	6., 30.
VI	23021 (99)	683 (3)	692	Four programs with experiments	7., 8., 9., 30.
VII	1441 (8)	111 (1)	68	Four programs testing functionalities	30., 32.

5.4 Threats to Validity

Several factors affect the validity of the evaluation findings:

- *Internal Validity* – Limited feature support and selective transpilation may overlook issues in unsupported or complex code;
- *External Validity* - The selected projects, while diverse, may not fully represent all C++14 applications, and testing was performed in a specific environment;
- *Construct Validity* - Reliance on existing unit tests assumes comprehensive coverage, which may leave some issues undetected;
- *Migration Strategies* - To address these threats, efforts were made to select a varied set of projects from different domains and to document all testing procedures and results meticulously in the CppUp repository. Future evaluations will aim to expand feature support and enable full project transpilation;
- *Developer Bias* – The evaluation was conducted internally; third-party reviews or independent test suites could improve objectivity.

6 CONCLUSIONS

This paper presented CppUp as a practical solution for modernizing legacy C++14 codebases to C++23, addressing 20 critical features and ensuring functional correctness. Systematic testing highlighted its reliability and utility, although limitations persist. CppUp currently lacks support for some deprecated or removed features and cannot process entire files in one pass due to the separate handling of preprocessing and translation.

Future enhancements will focus on extending parser capabilities, preserving comments, and expanding feature support to improve compatibility and facilitate more efficient and readable code modernization. The scalability, practicality, and detailed performance results of the tool will be reassessed after these improvements.

Integration of AI is also a potential avenue for further innovation, offering automated codebase analysis, identification of deprecated features, replacement suggestions, and validation of correctness through automated testing.

REFERENCES

Abramov, D., Redux [Online] [Accessed 28 Aug 2024] https://github.com/reduxjs/redux.

Antal, G., Havas, D., Siket, I., Beszédes, Á., Ferenc, R., Mihalicza, J., 2016. Transforming C++11 Code to C++03 to Support Legacy Compilation Environments. In: *IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM),* Raleigh, NC, USA, pp. 177-186. doi: 10.1109/ SCAM.2016.11.

ANTLR Another Tool for Language Recognition [Online] [Accessed 10 July 2024] https://github.com/antlr.

Bancila, M., 2024. *Modern C++ Programming Cookbook: Master modern C++ including the latest features of C++23 with 140+ practical recipes*, Packt Publishing, 3rd edition.

Bastidas Fuertes, A.; Pérez, M.; Meza Hormaza, J., 2023. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Applied Sciences*, vol. 13, 3667. doi: 10.3390/app13063667.

Chen, Z., Huang, K., Che, Y., Xu C., Zhang, J., Dai, Z., Mig, L., 2024. Extending OP2 framework to support portable parallel programming of complex applications. *CCF Trans. HPC,* vol. 6, pp. 330–342. doi: 10.1007/ s42514-023-00174-8.

Cooper, K.D., Torczon, L, 2011. *Engineering a compiler*, Morgan Kaufmann. San Francisco, 2nd edition.

GCC, the GNU Compiler Collection [Online] [Accessed 18 Nov 2024] https://gcc.gnu.org/

Gong, Y., Sbalzarini, I. F., 2017. Curvature filters efficiently reduce certain variational energies. *IEEE Transactions on Image Processing* vol. 26, no. 4, pp. 1786-1798. doi: 10.1109/TIP.2017.2658954.

Haxe programming language with a cross-compiler. [Online] [Accessed 16 Nov 2024] https://haxe.org/.

ISOCPP, 2024. *2024 Annual C++ Developer Survey "Lite"*, [Online][Accessed 14 Jan 2025] https://isocpp.org/ blog/2024/04/results-summary-2024-annual-cpp-developer-survey-lite.

ISO/IEC JTC 1/SC 22, 2014. *International standard ISO/IEC 14882:2014, Programming languages - C++.*

ISO/IEC JTC 1/SC 22, 2017. *International standard ISO/IEC 14882:2017, Programming languages - C++.*

ISO/IEC JTC 1/SC 22, 2020. *International standard ISO/IEC 14882:2020, Programming languages - C++.*

ISO/IEC JTC 1/SC 22, 2024. *International standard ISO/IEC 14882:2024, Programming languages - C++.*

Köppe T., 2018. Changes between C++14 and C++17. [Online] [Accessed 14 Jan 2025] https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0636r3.

Köppe T., 2020. Changes between C++17 and C++20. [Online] [Accessed 14 Jan 2025] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2131r0.html.

Lavavej, S.T., 2014. Removing auto_ptr, random_shuffle(), And Old <functional> Stuff. [Online] [Accessed 17 April 2024] https://www.open-std.org/jtc1/sc22/wg21/ docs/papers/2014/n4190.htm.

Ling, M, Yu, Y., Wu, H., Wang, Y., Cordy, J. R., Hassan, A. E., 2022. In Rust We Trust – A Transpiler from Unsafe C to Safer Rust. In: *IEEE/ACM 44th International Conference on Software Engineering: Companion (ICSE-Companion),* Pittsburgh, PA, USA, pp. 354-355. doi: 10.1145/3510454.3528640.

LLVM Clang. [Online] [Accessed 16 Nov 2024] http:// clang.llvm.org.

Lomas, A., 2014. Cellular forms: an artistic exploration of morphogenesis. In: *Proceedings of Special Interest Group on Computer Graphics and Interactive Techniques Conference*, SIGGRAPH'14, ACM. doi: 10.1145/2619195.2656282.

Marangoni, M.; Wischgoll, T., 2016. Paper: Togpu: Automatic Source Transformation from C++ to CUDA using Clang/LLVM. *Electron. Imaging*, vol. 28, pp. 1–9. doi: 10.2352/ISSN.2470-1173.2016.1.VDA-487.

Schnappinger, M., Streit, J., 2021. Efficient Platform Migration of a Mainframe Legacy System Using Custom Transpilation, In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Luxembourg, pp. 545-554. doi: 10.1109/ ICSME52107.2021.00055.

Schneider, L., Schultes, D., 2022. Evaluating Swift-to-Kotlin and Kotlin-to-Swift Transpilers. In: *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft),* Pittsburgh, PA, USA, pp. 102-106. doi: 10.1145/3524613.3527811.

Sneed, H. M., Verhoef, C. 2020. Cost-driven software migration: An experience report. *J. of Software: Evolution and Process*, vol. 32, no. 7, doi: 10.1002/ smr.2236.

Świniarski, A., CppUp [Online] [Accessed 14 Nov 2024] https://gitlab-stud.elka.pw.edu.pl/aderezin/ cppup_aswiniarski.

Taranov, K., Giromolo S., D., Hoefler, T., 2021. CoRM: Compactable Remote Memory over RDMA. In: *2021 International Conference on Management of Data*, SIGMOD'21, pp. 1811-1824. doi: 10.1145/ 3448016.34528.

TypeScript documentation -tsc, the TypeScript compiler. [Online] [Accessed 16 Nov 2024] https:// www.typescriptlang.org.

Xu, K., Zhang, G. L., Yin, X., Zhuo, C., Schlichtmann, U., Li, B., 2024. Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models. In: *2024 ACM/IEEE 6th Symposium on Machine Learning for CAD (MLCAD),* Salt Lake City, UT, USA, pp. 1-9. doi: 10.1109/MLCAD62225.2024.10740262.