Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 35–40
DOI: 10.15439/2022F308
ISSN 2300-5963 ACSIS, Vol. 30

Modern C++ in the era of new technologies and challenges—why and how to teach modern C++?

Bogusław Cyganek
[1]AGH University of Science and Technology, Poland
Al. Mickiewicza 30, 30-059 Kraków, Poland
[2]Academic Computer Center Cyfronet AGH
Ul. Nawojki 11, 30-950 Kraków, Poland
cyganek@agh.edu.pl

***Abstract*—Computers are one of the most important inventions in human history, and computer languages enable human-computer communication. Undoubtedly, C++ is one of the most important and influential in this group. Nevertheless, new technologies and related industry challenges place high demands on C++ and foster the development of new computer languages that meet new needs. For this reason, and thanks to the dynamically operating ISO standardization group, C++ is constantly updated while maintaining its backward compatibility. However, all this complicates and hinders not only the teaching of beginners but also the use by professionals. In this article, we briefly discuss the goals as well as proposed methodologies and techniques for teaching contemporary C++ in the age of new technologies and challenges.**

***Index Terms*—C++, modern technologies, compilers, teaching programming, computer science curricula**

I. Introduction

C++ is a multi-paradigm, imperative, procedural, functional, object-oriented, generic, and modular language invented in early 1980s and further developed by Bjarne Stroustrup [5][10] [29]. Since 1991 standardization of C++ is supported by the ANSI International Organization for Standardization (ISO), with the latest standard version published in December 2020 [21]. With performance and efficiency in mind, C++ extends and is compatible with the C programming language, while to incorporate the object-oriented and abstraction mechanisms it draws from Simula. This hybrid approach has proven extremely useful over the years, especially in such domains as systems programming, embedded systems, resource constrained platforms, large computing and simulation libraries, machine learning & artificial intelligence (ML/AI) and many others.

Nevertheless, there are industries such as web applications that favor the development of other languages as well. While the popularity rankings of programming languages are in some ways superficial and may be misleading, they provide some insight into future trends in the IT industry and can help students decide which language they want to learn. From these the TIOBE Programming Community index shows the popularity of programming languages based on 25 search engines [16]. At its top are Python, C, Java, and C++, which together are well ahead of the others, as shown in Fig. 1. In the last two years, Python and C have swapped between 1st and 2nd places in the ranking. Also C++ is gaining in popularity and tends to surpass Java.

Aug 2022	Aug 2021	Change	Programming Language	Ratings	Change
1	2	▲	Python	15.42%	+3.56%
2	1	▼	C	14.59%	+2.03%
3	3		Java	12.40%	+1.96%
4	4		C++	10.17%	+2.81%
5	5		C#	5.59%	+0.45%

Fig. 1 An excerpt from the TIOBE list of the top-ranked programming languages in 2022 (from [16]).

On the other hand, in the PopularitY of Programming Language Index (PYPL), which shows how often language tutorials are searched on Google, C/C++ are ranked 5th together (Aug. 2022) [17]. However, neither of the above indexes is about the best programming language or the language in which most lines of code have been written.

A detailed analysis of various languages and their applications is far from the scope of this paper, nevertheless we can observe that while scripting tasks surely fall into the realm of Python, and web development for Java, then vast majority of high performance applications falls into the domain of C/C++.

The latter are also the only languages in this group that compile their code directly into the machine language. Although, there are contenders such as Rust and the recent Carbon [11][12], C++ endowed with hundreds of libraries, tools, and many years of experience, and a superset of C in a sense, *is and will probably be the most important and productive language today*, especially for large and performance demanding systems. Therefore, C++ is surely worth learning. However, the more extensive the specification of modern C++ becomes, the more critical the requirement to properly teach modern C++ to new generations of programmers becomes. In this article, we tackle this issue in an attempt to shed more light on why and how to teach modern C++.

There are relatively large Internet resources [30][10][13] and literature [27][20][23] about C++ and its features. However, when it comes to teaching modern C++, the situation is not bright. There are only few online presentations [2][3][4], web services and books to recommend [28][29]. Nevertheless, even these are a bit dated considering new C++17 and C++20 standards. To fill this gap the new book was written, *Introduction to Programming with C++ for Engineers*, which was published by Wiley-IEEE Press in 2021 [6]. It contains teaching materials, from elementary to advanced level, intended for the three-semester study cycle. Based on this, this article provides an overview of methodology and techniques for teaching modern C++.

It is worth mentioning that the problem of teaching and disseminating knowledge about modern C++ also found wide interest in the language committee. In this context, the SG20 group arose, whose aim is to prepare and provide guidelines for content to be covered by C++ courses [31]. Their main document is a resource for instructors to assist in the preparation of C++ courses in a variety of environments, including universities, colleges, and industry.

The rest of the paper is organized as follows. An overview of the methodologies and techniques of teaching the C++ language in presented in Section II. It is organized into four subsections. Section III provides scenarios for the different levels of C++ learning. Section IV discusses the role of good examples in the teaching process. Section V deals with the issue of teaching for real life challenges. The paper ends with conclusions in Section VI.

II. An Overview of Teaching Methodologies & Techniques

In this section the basic methodologies and techniques for teaching modern C++ are outlined. The main issue is to list the most important steps in class preparation and to focus on the most important factors.

A. Preparing for Teaching

First, there are some key factors to consider before starting your class. The following is a list of them:

- Get to know your students – what are their backgrounds, what are their motivations, whether they are kids or students of electronics and telecommunication, computer science students, or students of non-technical faculties (biology, humanistic, etc.); or professionals who want to expand their skills in modern C++? What have they already learned, math, python, basics of computer science?
- Organize your classes well – individual or group work (some activities such as lectures can be for a group, but some – such as tutorial – should be individual), group sizes, etc. Have a plan but actively respond to students' progress and expectations, have close contact and react actively, similarly to the *agile* methodology for software development. But also control the attitude and experience of other fellow teachers in the group (in many universities often the lecturer and laboratory teacher are different people).
- Plan your time – how many hours for a lecture, for a lab, for joint work, and for an individual project. Consider time for individual consultations.
- Organize the class work well – consider exercises for personal work as well as projects for *team work*.
- Teaching materials – students have access to various sources, but they rely on your opinions, therefore the correct selection of book(s), internet materials, video(s), etc. is very important.

Certainly, these are only propositions based on many years of our observations and conducted classes. However, for different groups and teachers, the list and importance of each factor may differ.

B. Choice of the Vital Language Features – the 20/80 Rule

What works well in our 25 years teaching experience is getting the right preparation and then focusing on the most important and productive features of the language at the given teaching stage that allow students to quickly comprehend and become proficient in basic programming techniques. As a result, it allows the students to create useful and well-organized programs as quickly as possible. The choice of features can be arbitrary, but is best if these are based on the experience of the teacher(s). As we have noticed, for this purpose *the Pareto 20/80 principle* is worth considering [14][6].

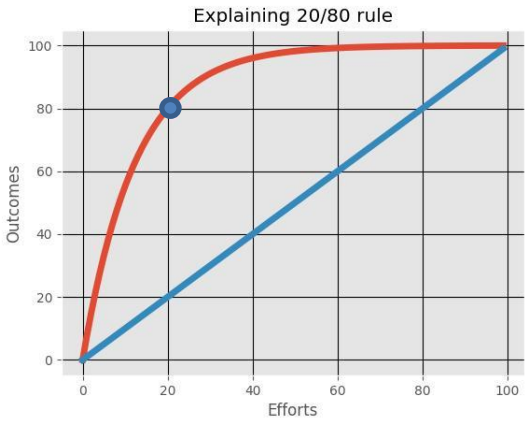

Fig. 2 The 20/80 rule states that many activities are not evenly distributed and some contribute more than others. This idea can be used to prepare 20% of the most important C++ features for the students' classes.

The 20/80 paradigm, known also as the Pareto rule or law of the vital few, states that in many life situations, 20% of causes is responsible for roughly 80% of consequences, or results, as shown in Fig. 2. This heuristic observation was probably first noticed by the mentioned Italian economist Vilfredo Pareto, who noted that at that time 80% of the land in Italy was owned by 20% of the population. Interestingly, this can also be observed in computers, which is usually manifested that 20% of bugs contribute 80% of crashes, or that 80% of the CPU time is spent on 20% of the code, etc.

Hence, the idea is to use this rule to prepare the most important 20% of features to be taught in the beginning. This approach can result in much better productivity and allows students to faster reach the level of solid understanding of basic programming constructs and techniques, compared e.g. to the linear approach, as shown in Fig. 2.

C. The Spiral Development Model

As originally proposed by Boehm [1][15], the spiral model of software development is associated with iteratively repeated processes while managing risk for its active reduction, as shown in Fig. 3.

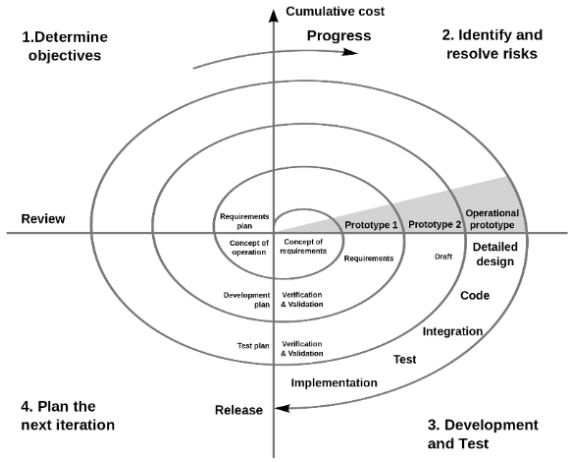

Fig. 3 The spiral model, originally proposed by Boehm for development of software, can be also applied to the C++ teaching process (from [1]).

However, it appears that the C++ teaching process is in many ways similar to this spiral model. Namely, many topics are, and should be, repeated with wider scope and level of details. Risk management in this case means strict control of the students' absorption of previous material before presenting an advanced version of a topic. This also applies to the gradual increase in the complexity of student developed projects.

III. EXEMPLARY TEACHING SCENARIOS

Some scenarios for teaching modern C++ are discussed here. Assuming classes organized in the semester periods (14/15 weeks per semester), and organized in the form of a lecture per week, a laboratory per week for the half of the semester, as well as the student's own project and consultations for the remaining part of the semester, the following scenarios are presented for the entire three-semester C++ teaching cycle:

1. Introduction to programming with C++ (the beginners program), followed by basics of C++.
2. Object-oriented design & programming with C++, followed by advanced memory management.
3. Advanced C++ concepts, followed by basics of parallel programming.

A possible organization of classes in the form of a state diagram is shown in Fig. 4. Each of the three semesters consists of *two building states* – the idea here is that the second state in a semester is optional, i.e. it is undertaken if there is enough time and the group have achieved good results in the first state.

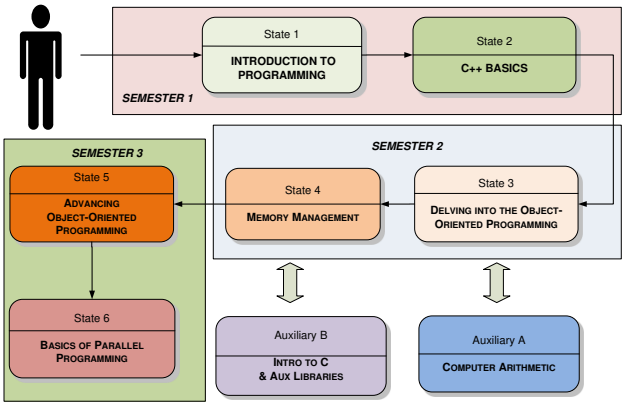

Fig. 4 Possible C++ teaching scenarios organized for the three semesters. Each semester consists of two stages – compulsory and optional, which is carried out if there is enough time and the group has demonstrated significant progress in learning. Also visible are auxiliary topics "A" for computer arithmetic and "B" for low-level features, such as the C programming language and introduction to additional libraries.

These are supplemented with the auxiliary states, which can be included to the program of teaching semesters depending on the needs and progress of the students. Supporting topics include: "A" computer arithmetic and "B" introduction to programming in C and a supplemental introduction to using libraries such as QT, FLTK, OpenGL, OpenMP, OpenCV, etc., depending on the students' needs and the profile of their faculties.

However, before providing some more concrete lists of features for each teaching scenario, let's highlight the following issues and hints that should be considered:

- Well define the main goals of the classes.
- For each semester well define a minimal set of C++ features to be acquired by students; for this purpose the 20/80 rule can be applied.
- Throughout the term: stick to the developed teaching plan (the curriculum) but actively respond to students' progress – this resembles the *agile* concept, applied to the teaching process.

Not surprisingly, the aforementioned teaching scenarios follow chapter layout in the book [6]. The more detailed teaching scenarios are outlined in the following subsections.

A. Scenario for Beginners

Following the plan outlined in Fig. 4 let's analyze a possible minimal set of C++ features. This can be defined as follows.

1. Introduction to the computer API and the basic C++ development tools (editor, compiler, linker, IDE, etc.).
2. The `main` function.
3. Minimal libraries (`#include`), `using` directive.
4. Printing texts `std::cout`.
5. Defining and *initializing* variables: `int` and `double` (explain the difference).
6. Entering values to the variables `std::cin`.
7. Conditional statement `if` and how to provide a logical condition.
8. `std::vector`
9. `std::string`
10. The loop: "classical" `for` and "range" `for`

Especially for beginners it is important to provide diagrams and ready recipes for project organization, tools, build process, particularly compiling and dealing with compilation errors as well as basics of debugging. Active support from the teacher is essential at this stage. A possible diagram of an exemplary program in the single `main` function is shown in Fig. 5.

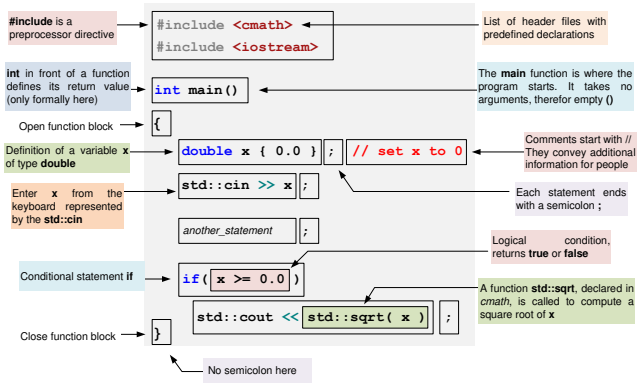

Fig. 5 A diagram showing basic constructions of a C++ program for absolute beginners (from [6]). Many projects for beginners can be written on the canvas of a single `main` function.

Although very simple, the code based on a single `main` function can be used successfully in many beginner projects.

B. Scenario for C++ Basics

A possible scenario for teaching the basics of C++ may include the following topics.

1. The most common built-in data types, their applications and initialization.
2. Code debugging techniques.
3. Basic members and applications of `std::vector`.
4. A matrix as a `vector` of `vectors`.
5. Basics of `std::string`.
6. `auto` and when to use it.
7. Common standard algorithms (`std::copy`, `std::find`, `std::generate`, `std::accumulate`, etc.).
8. Structures as data containers with `struct`.
9. References.
10. Statements (the role of braces).
11. Functions (argument passing, recursive, lambdas).
12. Intro to (separate) classes: `struct` vs `class` + constructor and member functions.
13. Basic of software testing.
14. Operators.

The beginners and basics scenarios constitute teaching material for the 1st semester (Fig. 4).

C. Scenario for Object-Oriented Programming in C++

A natural follow up is introduction to the OOP domain with C++. At this stage a possible list of topics can look as follows.

1. Intro to OOP.
2. Anatomy of a class (e.g. extended matrix class).
3. Right references.
4. Classes with all special functions – move semantics explained (e.g. extending matrices into tensors).
5. Templates and generic programming (functions, classes, member templates).
6. Virtual mechanism.
7. Some design patterns (e.g. wrapper, handle-bode/bridge, proxy).
8. Memory management (RAII, `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`).

The virtual mechanisms and polymorphism should be shown on class hierarchies. However, the more complex are postponed to the advanced level, as discussed in the next section. Nevertheless, the introduction to the OOP and topics of memory management constitute material for the 2nd semester.

D. Teaching Advanced Topics of C++

After completing the OOP and memory management parts of the course, the last stage can be coined "advanced C++". However, it is not less difficult to define what actually should be included and how to teach such advanced concepts. Nevertheless, a possible list of topics can be formed as follows.

1. Designing class hierarchies.
2. C++ filesystem.
3. Forward/universal references.
4. Regular expressions (`std::regex`).
5. Graphical user interface (various libraries, QT, MFC, FLTK, …).
6. System clock and time measurement (`std::chrono`).
7. Intro to functional programming and the `std::ranges`.
8. Intro to expression parsing – the interpreter DP, building the syntax trees, composite DP, visitor DP.
9. State machine pattern.
10. Advanced generic programming techniques.

11. Basics of parallel programming (`std::par`, `std::thread`, OpenMP).

Certainly, all the aforementioned teaching scenarios are just simple propositions [6]. They can be easily adjusted to best suit the level and needs of the students as already discussed.

IV. The Impeccable Role of Good Examples

As it is not possible to teach swimming without entering the water, it is also not possible to teach programming without writing code and developing projects by students. Hence, the role of good code examples cannot be overestimated. However, let's consider what factors should be taken into account when preparing code examples for teaching. A good example of code should be:

- comprehensible,
- not too long,
- touch on 'real' problems,
- can serve as a starting point for student's project; i.e. it can be used in an incremental development and the spiral model in action.

The examples are very important because not only do they illustrate some code concepts, but they provide "thought patterns" that the developer evokes and then modifies while working on a similar problem.

There are different ways students can use the code examples, for instance:

- Compile, make it run, debug to observe the results.
- Re-type an example and make it run.

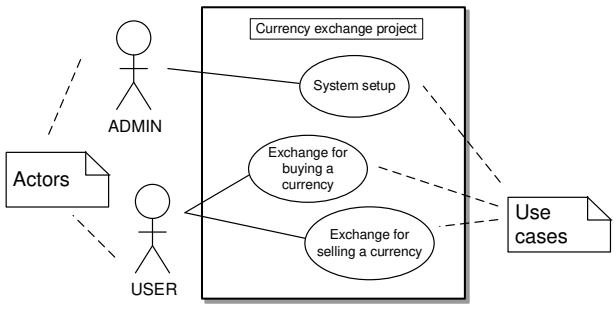

Fig. 6 Teaching software development is about more than showing C++ code. The entire process should be presented, starting e.g. with the UML diagrams and the complete development process, from design, till code unity tests and deployment. Here is a UML use case diagram of the currency exchange exemplary project (adopted from [6]).

- Don't use the example – create your own version, instead; then compare.
- Use as a 'startup' for your own project/example.

It is also important for educators to realize that teaching programming should encompass *the deeper context of software development*. This means that instead of showing only parts of C++, a teacher should present the entire development process, starting with problem analysis, resulting UML diagrams such as in Fig. 6, software design, code unity tests, deployment, etc.

However, we'd like to emphasize that not less important is the process of consultations between the student and her/his teacher. Such a code refinement process enables comprehension of proper coding techniques.

V. Teaching for the Real-Life Challenges

The world did not start today, and there are millions of lines of the legacy code that still need to be maintained, deployed, refactored or extended. Therefore, teachers should ask themselves questions such as how to prepare students to cope with the existing code, as well as how to respond to the expectations of employers. The problem is even deeper – there is an ongoing discussion on mismatch between computer science curricula and industry needs [22][24][25][26][32], which generally is out of the scope of this paper. However, when confining this to teaching modern C++ the teachers should be aware of the *what* to address and *how* to go about, as well as of the *short time span* issues.

Returning to the task of teaching modern C++ in the light of the legacy code, we need to introduce the lower-level constructions of C++ (such as pointers, unions, raw memory/string operations, etc.), sometimes also at least the basics of C. Certainly, the set of these low-level features depends on the needs of the students. However, the most important are the proper moments in the teaching scenarios when these low-level features are taught. That is, low-level features should be introduced *after* instilling good programming habits in using *modern* features of C++, not the other way around.

The low-level features will be inevitable when preparing students to work with operating systems such as Linux or FreeRTOS, or to use many common libraries, such as OpenMP, OpenCV, OpenGL, QT or games with Unreal Engine, etc. But even to explain at some point what is `int main( int argc, char ** argv )` the teacher has to face how to introduce the pointers. So the question is not "if" but "when".

VI. Conclusions

The paper contains a short overview of the challenges related to promoting the interest in modern C++, as well as in teaching modern C++ to various groups of learners in the era of new technological challenges. The subject is very wide and we only scratched the surface.

Every three years, the C++ standardizing committee, supported by thousands of enthusiasts, publishes a new specification for the language. Thanks to this, it gained dozens of new features, which makes it very efficient and effective. However, such a dynamically changing environment also raises some problems especially in terms of the *stability* of language features as well as their presentation and acceptance from the world C++ programming community.

This also makes teaching modern C++ a real challenge. Accordingly, this paper introduces some teaching methodologies and techniques, such as the 20/80 principle, as well as some modifications to the software development

methodologies, such as the spiral model and agile approach, which were brought into the teaching domain. Further on, the overviews of some teaching scenarios for different groups of learners were provided. More detailed versions are available in the book *Introduction to Programming with C++ for Engineers* [6], whereas the code examples and additional materials are available from the Internet [8][9].

In the end, the following list summarizes the main postulates and guidelines for effective teaching of modern C++:

- As C++ programmers we are all students and many of us are, or become, teachers.
- When teaching, get to know your students, get to know their needs and wishes, then organize the classes well.
- At each stage think about selecting the appropriate C++ features for teaching, keeping in mind *the 20/80 rule*.
- Provide good and practical examples! They constitute "mental patterns" for your students.
- Keep things simple and in right order – but be *agile* and actively react to students' progress.
- Teach in a repeated way, gradually introduce more advanced concepts and techniques, raising the level – apply the *spiral* development model.
- Teach not only C++ itself, but C++ across *the entire computer science framework*: show steps of software design, UML, data structures, algorithms, design patterns, development tools, software testing, etc.
- Do not forget about the programmers/companies reality, the legacy code, etc. Provide information on lower-lever features, teach basics of C if necessary, but at the right time.
- Pay attention to the self-education, always improve the skills of not only your students, but also yourself and your team, watch/participate in lectures, conferences, read books, etc.

With the increasing demands on high performance systems, the embedded world moving from C to C++, the revolution in big data, parallel computations, etc., undeniably C++ is, and probably will be, the most powerful modern computing language for many years to come. Therefore, there is no doubt that *C++ should be taught to a wide range of students*, especially of technical faculties. However, this should be well organized as *one of the important topics* in wisely prepared computer science curricula, containing *a combination of various programming languages*, that respond to the needs of industry in the era of new technologies and challenges.

As a concluding remark let's remember that: *The quality of the software of the future depends on the quality of education today*.

ACKNOWLEDGMENT

The author expresses his gratitude to Prof. Dominik Ślęzak for his invitation and encouragement to write this paper.

The author also commends Wiley-IEEE Press for the 2021 Wiley-IEEE Press Professional Book Award for the book *Introduction to Programming with C++ for Engineers* (https://ieee-press.ieee.org/wiley-ieee-press-awards/) and for financial support in participation in FedCSIS'22.

REFERENCES

[1] Boehm, B: Spiral Development: Experience, Principles, and Refinements. Special Report. Software Engineering Institute, 2000.

[2] CppCon 2017: Bjarne Stroustrup "Learning and Teaching Modern C++" – YouTube https://www.youtube.com/watch?v=fX2W3nNjJIo

[3] CppCon 2015: Kate Gregory "Stop Teaching C" – YouTube https://www.youtube.com/watch?v=YnWhqhNdYyk

[4] CppCon 2018: Christopher Di Bella "How to Teach C++ and Influence a Generation"—YouTube https://www.youtube.com/watch?v=3AkPd9Nt2Aw

[5] C++ Wikipedia: https://en.wikipedia.org/wiki/C%2B%2B

[6] Cyganek B.: Introduction to Programming with C++ for Engineers. Wiley-IEEE Press, 2021.

[7] Gurcan, F., Kose, C.: Analysis of software engineering industry needs and trends: implications for education. International Journal of Engineering Education, Vol. 33, pp. 1361-1368, 2017.

[8] https://home.agh.edu.pl/~cyganek/BookCpp.htm

[9] https://github.com/BogCyg/BookCpp

[10] https://cppreference.com

[11] https://thenewstack.io/google-launches-carbon-an-experimental-replacement-for-c/

[12] https://9to5google.com/2022/07/19/carbon-programming-language-google-cpp/

[13] https://stackoverflow.org

[14] https://en.wikipedia.org/wiki/Pareto_principle

[15] https://en.wikipedia.org/wiki/Spiral_model

[16] https://www.tiobe.com/tiobe-index/

[17] https://pypl.github.io/PYPL.html

[18] https://www.devjobsscanner.com/blog/top-8-most-demanded-languages-in-2022/

[19] History of C++ https://en.cppreference.com/w/cpp/language/history

[20] Josuttis N.: C++17 - The Complete Guide: First Edition, 2019.

[21] JTC1/SC22/WG21 - The C++ Standards Committee – ISOCPP, 2022. https://www.open-std.org/jtc1/sc22/wg21/

[22] Lawlis P.K., Adams K.A.: Computing Curricula vs. Industry Needs: A Mismatch. Proc. 9th Annual ASEET Symposium, pp. 5-19, 1995.

[23] Meyers S.: Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, 2014.

[24] Moreno A.M., Sanchez-Segura M-I, Medina-Dominguez F., Carvajal L.: Balancing software engineering education and industrial needs, Journal of Systems and Software, Volume 85, Issue 7, pp 1607-1620, 2012.

[25] Oguz, D., Oguz, K.: Perspectives on the Gap Between the Software Industry and the Software Engineering Education. IEEE Access, Vol. 7, pp. 117527-117543, 2019.

[26] Paprzycki M., Zalewski J.: CS II: An Applied Software Engineering Project. The Journal of Computing in Small Colleges, Vol. 12, No., pp. 47-52, 2, 1996.

[27] Stroustrup B.: The C++ Programming Language, Addison-Wesley, 2013.

[28] Stroustrup B.: Programming: Principles and Practice Using C++, 2nd Ed., Addison-Wesley, 2014.

[29] Stroustrup B.: A Tour of C++, Addison-Wesley, 2018.

[30] Stroustrup B., Sutter H.: C++ Core Guidelines, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[31] SG20 (ISO C++ Study Group on Education): Guidelines for Teaching C++, 2022. https://cplusplus.github.io/SG20/latest/

[32] Waks S., Frank M.: Engineering Curriculum versus Industry Needs – A Case Study. IEEE Tr. on Education, Vol. 43, No. 3, pp. 349-352, 2000.