Introducing C++20 Modules in First-Year Computer Science: A Pedagogical Case Study

Siphesihle Sithungu
University of Johannesburg, Johannesburg, South Africa

siphesihles@uj.ac.za

Abstract: This paper presents a case study on introducing C++20 modules into a first-year computer science curriculum to expose students early to modern modular programming. In line with constructivist learning principles, modules were introduced as bonus content in the first semester, allowing students to learn through hands-on practical tasks. The integration faced technical challenges, especially around compiler compatibility, where initial attempts using MSYS, LLVM, and Clang were unsuccessful. Eventually, GCC 14.2.0 was adopted to support C++20 modules. A scaffolding approach was used to support student learning, starting with a virtual programming lab and gradually transitioning to local lab environments. Supplementary resources, including tutorial videos, were developed to help students move from beginner tools like Code::Blocks to more advanced editors such as Visual Studio Code. The study highlights the benefits of project-based and technology-enhanced learning, showing that students responded positively to the modern C++ content. This paper offers insights into the practical implementation of contemporary C++ features in introductory courses and contributes to ongoing discussions about effective programming pedagogy.

Keywords: Modern C++, Curriculum Innovation, Computer Science Education, Technical and Pedagogical Integration, First-Year Curriculum

1. Introduction

C++ is one of the most used programming languages in the history of computer programming due to the expressive ability it provides to computer programmers and software engineers. The applications of C++ span a wide variety of domains, such as systems programming (Chan, 1996), game development (Sutherland, 2014; Mack & Ruud, 2019), back-end development (Reddy, 2011) and compiler construction (Holmes, 1994). Python is an example of programming languages we use daily that were written in C or C++ (Chun, 2001). The C++ language – initially called *C with Classes* – was primarily incepted as an object-oriented take on its predecessor, C, by Stroustroup, and it was first publicly released in 1983 (Stroustrup, 1996). C++ - along with C- is also the language used to write the influential Compute Unified Device Architecture (CUDA) (Li, et al., 2019), designed for programming Graphics Processing Units (GPUs) (Sanders & Kandrot, 2010).

Given the influence and features of C++, it is unsurprising that it is one of the most taught programming languages at universities globally in undergraduate computer science courses. A significant number of universities teach C++ as part of their undergraduate or graduate programs (Tan, et al., 2014), which is a testament to the language's importance in both foundational concepts and advanced technical applications. Our university is no different in its appreciation of the importance of introducing computer science students to core programming principles through C++.

As with most languages, the C++ programming language is continuously evolving to remain a relevant choice for computer programmers. One of the main improvements introduced in C++20 was the ability to achieve modular programming in C++ using modules (Szalay & Porkoláb, 2025), which are expected to replace header files. Modules were introduced to address critical limitations of header files, which include order sensitivity, slow compilation, limited optimisation and exposure to internal details such as macros. It is worth noting that header files have been a building block of C++ programs since the language's inception and they were also a part of C, which made the introduction of C++20 modules a critical turning point in the evolution of the language.

Given the evolutionary significance of modular programming in C++, it is no surprise that none of the major compilers at the time of writing this paper, such as GNU Compiler Collection (GCC), LLVM (Low-Level Virtual Machine) Clang and Microsoft Visual C++ (MSVC) have fully implemented modular programming as prescribed by ISO C++ (Szalay & Porkoláb, 2025). This situation presents an interesting challenge for educational practitioners: ensuring that students remain exposed to the most up-to-date and current developments in C++ in a way that accommodates ongoing compiler support for C++20 modules.

To enable students to build C++ knowledge through modular programming, a constructivist approach to teaching and learning (Von Glasersfeld, 2012) became a natural requirement in order to create a conducive environment for active learning (Felder & Brent, 2009).

This paper presents a case study on the integration of C++ 20 modules into a first-year computer science curriculum, with the main focus being on the pedagogical strategies followed, the technical challenges encountered and student engagement. The main contributions of this paper are as follows:

- The paper presents a real-world case study of how modular programming in C++ can be introduced to first-year computer science students, where modules are offered as bonus content in the first semester and become compulsory in the second semester. This contribution advances the conversation around modernising programming pedagogy in line with the evolving C++ standard.
- The paper documents obstacles encountered in choosing the right tooling for modular programming in C++ within an educational environment. A discussion is provided on navigating real-world compiler ecosystem challenges to provide insights that others can use to implement modern C++ features in constrained environments.
- Finally, the paper reflects on anecdotal indicators of healthy adoption and student engagement with C++20 modules, which contribute to ongoing efforts to modernise programming education.

The rest of the paper is structured as follows: Section 2 describes the general course design, reflecting on the strategies used to teach modular programming in C++, the challenges encountered and solutions that emerged. Section 3 discusses the results of having introduced C++20 modules, focusing on three main aspects: (1) student engagement, (2) impact on learning outcomes, and (3) technical outcomes. Section 4 offers a discussion that reflects on the outcomes of the project by analysing the pedagogical approach, the challenges and lessons learned and student feedback.

2. Course Delivery

This section discusses the pedagogical approaches employed to teach modular programming to first-year, first semester computer science students to provide a clear picture of how the course was designed and delivered. Thereafter, the challenges faced throughout the semester are reflected upon, including how each challenge was addressed. Most challenges experienced were related to tooling and making the content accessible to students of varying abilities.

2.1 Pedagogical Approaches

In order to successfully introduce C++20 modules, it was important to lean towards a constructivist teaching approach, which views learning as an active process requiring students to engage with their surroundings and experiences in order to build knowledge (Von Glasersfeld, 2012). Constructivism discourages passive learning and encourages active learning. Adopting such a philosophy provided benefits in this context given the practical nature of computer programming and the fact that a new way of writing C++ programs was being introduced.

The transition from traditional header file-based C++ programs to modules should not be underestimated given that header files have been the only approach for creating programmer-defined libraries in C++. It was therefore important to encourage as much “hands-on” learning as possible from our students given that a fundamentally different approach to writing programs was being introduced in our course. Being the very first time that modular programming was being taught in the first year meant that our students could not consult course materials from previous years for revision purposes. Therefore, they had to be given as many opportunities as possible to develop hands-on experience in learning how to create libraries using modules.

Following a constructivist approach was not sufficient. We also had to ensure that students were not overwhelmed by the content. This was achieved by gradually introducing concepts through a scaffolding approach (Gibbons, 2002), where the ability to create modular C++ programs was only made available on the department’s Virtual Programming Lab (VPL). The VPL enabled students to create source files, write source code and execute it without knowing how to actually compile it, thus scaffolding away the implementation details of modular programs in C++.

Over time, after determining that students were comfortable with the program structure and knew how to create the appropriate source files, we then transitioned to teaching them how to compile a C++20 program containing modules via the command line interface (CLI). Even more so, to ensure that students were not overwhelmed with new information, the use of modules was only reserved for bonus assessments, allowing them to continue to use header files for the main assessments. This allowed students to engage with C++20 modules at their own pace, while rewarding their attempts with bonus marks.

As mentioned previously, active learning was an essential component of the pedagogical strategy as it enabled students to engage with the material through hands-on coding, debugging and problem-solving. Active learning directly supports the constructivist philosophy adopted for this particular project. The various tools that were made available to students in enabling them to engage with modular programming in C++ supported technology-enhanced learning as they facilitated a modern learning environment. Traditionally, over many years, our course had used CodeBlocks (Soto & Figueroa, 2018) as the official Integrated Development Environment (IDE). However, the introduction of modules introduced the opportunity to shift attention towards the VPL as well as Visual Studio Code (VS Code). Finally, because students had to create solutions to practical assignments, even though they were part of the bonus component of the course, in doing so, it created an environment for project-based learning.

2.2 Challenges and Solutions

The introduction of C++20 modules was met with numerous technical and administrative challenges, both of which will are discussed separately. The challenges faced during the first semester provided invaluable insights into how the delivery of the content could be better structured in the second semester.

2.2.1 Technical Challenges

One of the foremost technical challenges faced was choosing the appropriate compiler that supports the C++20 standard. Various C++ compilers exist, and some are more niche (or specialised) than others. The most mainstream C++ compilers that are used extensively in both academia and industry are the GCC, LLVM and MSVC. Given that the CodeBlocks IDE has always been used for the course, and CodeBlocks ships with GCC by default, GCC has always been the *de-facto* compiler for the course. However, the latest version of CodeBlocks (20.03) supports up to C++17. This meant that students would have had to create source files manually and use the command line interface (CLI) to compile and run their programs. Another alternative would have been to choose a different editor or IDE, such as Visual Studio Code. However, syntax highlighting in VS Code highlights module-related code as erroneous, which would have added frustration to first-year students.

C++20 support began in GCC 13, and it has been on technical specification[1] until now (we are currently on GCC 14.2.0 as of the writing of this paper). This meant that not all of the C++20 standard was supported by GCC and not every feature/functionality specified in ISO C++ would work. Another issue we experienced was that the GCC version installed on the VPL instance was GCC 13, which certainly introduced a layer of complexity given the difference in C++20 support, as described earlier. It would have been risky to create an entirely new VPL instance with GCC 14 installed given that our VPL is not only used by our computer science first-year students, meaning that other courses might have been affected by the migration.

It is also worth noting that we experimented with the other compilers, such as Clang and MSVC, but both of these alternatives also had limited support for C++20 modules. Given that the main part of the course already makes use of GCC, it was sensible to stick to GCC to prevent confusion or frustration for students. Using the same compiler toolchain meant that we did not have to teach a new compiler to students wanting to engage with the bonus content. Our only goal was to explain to them that the current version of GCC does not yet fully support modules and highlight quirks they might encounter when writing modules in C++. This approach eliminated a significant amount of complexity and ensured that the delivery of the content was straightforward.

To summarise, given that the current version of CodeBlocks does not support C++20 modules, our students had two options: (1) use the dedicated VPL coding area for bonus content or (2) use any text editor to write C++20 programs and compile from the CLI. The second option gave students a more solid understand of the C++ compiler ecosystem including the compilation and linking stages.

Another challenge encountered when delivering the bonus content was to decide on an extension for the source file containing a module. Different conventions exist, as is typically the case in the C++ community, depending on the compiler toolchain. It seems to be the consensus within the GCC community to use the .cxx file extension. Although we initially went with .cxx, we eventually settled with .cpp to simplify the compilation process on the VPL. While the literature might suggest otherwise, we found that GCC 14 does not support the use of separate

[1] Technical specification means that the support for modules is still experimental.

interface and implementation units, which further justified the choice of using the .cpp extension prototypes and implementations were written in the same file.

2.2.2 Administrative Challenges

To successfully teach modular development in C++, new learning material needed to be created for the course. Since modules only formed part of the bonus content, only tutorial videos were created and hosted on our department's YouTube channel, and each video tackled a specific concept. Each video was short and objective, addressing a specific aspect of working with modules. The second administrative challenge was incorporating the bonus content into weekly practical assignments and the practical semester test. To maintain consistency, a separate bonus section was added to each practical assignment with a fixed number of marks.

The practical test is the most important assessment students complete during the semester. It is a three-hour closed-book assessment, completed at our practical lab venues in one sitting, and it contributes 65% towards the semester's mark. Given the critical nature of this assessment, it is standard practice to incorporate a bonus section allowing students to earn extra marks to maximise their chances of success. This year, modules were included as part of the bonus content. However, to make the bonus section accessible to a wide variety of students (even those who did not understand modules), the bonus section also included using functional programming concepts in C++ (i.e. lambda expressions, passing functions as parameters to other functions, etc.)

Another administrative challenge encountered was with regards to marking. Given the large number of students enrolled for our course, we need the help of student assistants to mark weekly practical assignments and semester tests. Given that this was the first time C++20 modules were being taught, it meant that our student assistants, who completed the course in the previous years, were not familiar with C++20 modules. To address this challenge, marking assistants were instructed to watch the same tutorial videos provided to students so that they not only familiarised themselves with the concepts, but were also aware how the concepts were taught to the students. This approach ensured that the marking assistants' expectations (when marking) were in line with what was taught.

Other administrative challenges were expected, such as ensuring the availability of bonus coding areas for weekly practical assignments, the practical semester test and final exam. However, some administrative challenges directly manifested from unexpected technical challenges. An example of a quirk that was discovered during the course was when passing a lambda expression as a parameter to a function defined inside a module when using GCC 14. We found that, if a function takes a lambda expression as a value parameter and passes it to another function as a value parameter, the compiler will see the lambda as being defined more than once (which was obviously not the case), resulting in a compile-time error. A workaround here was to have one of the functions accept a lambda expression as a reference (or a const reference to avoid modifications) parameter. Such arising quirks had to be clearly communicated to students.

The administrative challenges arising from the introduction of C++20 modules in our first-year computer science course provided a significant number of valuable lessons, not only about how to teach the most cutting-edge features of a programming language, but also about the evolution of modern C++ itself.

3. Results

The work undertaken to introduce modules in first-year computer science resulted in several outcomes, which are presented in this section. First, a reflection is made on how students engaged with modules as part of bonus content. Engagement was tracked by monitoring activity logs on the VPL as well as directly observing the number of students who attempted the bonus sections of practical assessments. Thereafter, an assessment of the impact of bonus activities on learning outcomes is briefly conducted by investigating how students who attempted bonus activities performed overall in the course.

3.1 Student Engagement and Adoption

To evaluate whether students were engaging with the bonus content, we tracked the number of attempts for the bonus content of practical assignments based on submissions, as well as the activity logs on the VPL. Although only a small fraction of the students were interested in the bonus content, those who were interested consistently attempted bonus questions. Activity logs on the VPL also showed that certain students consistently used the coding area dedicated to bonus questions. Figure 1 is a chart reflecting the usage of the bonus coding area by one of the students.

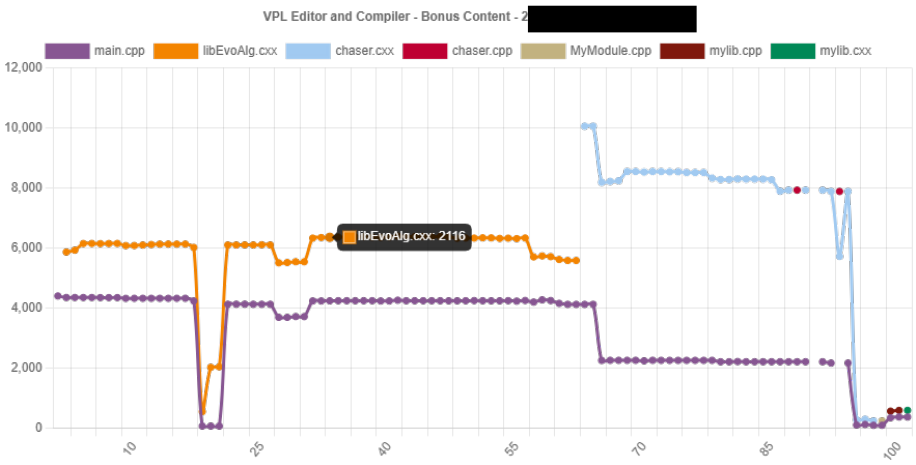

Figure 1: VPL activity log for one student. Student details were removed to preserve anonymity.

The different line colours reflect changes made to corresponding source files (source files are listed at the top of the chart). This type of chart depicts student interaction with the coding area – and therefore – the bonus content. Being able to track engagement in this manner also enabled observing whether students who engaged with bonus content eventually succeeded in the course (and if they did, what their overall performance was). The bar chart shown in Figure 2 shows the total number of hours for which the same student observed in Figure 1 used the bonus coding area. As can be seen from the top of the chart, the student spent a total of 3.2 hours using the bonus coding area.

Figure 2: A bar chart showing a breakdown of the number of hours (y-axis) the student used the bonus coding area per working session (x-axis).

Students received a total of eleven practical assignments throughout the semester and three of those assignments involved the use of C++20 modules as a bonus question. We shall refer to these assignments as P_A, P_B and P_C. The course had a total of 607 registered students, and the number of students who attempted the bonus sections of each of the three assignments was as follows:

- P_A: 85 students (14% of the class)
- P_B: 42 students (7% of the class)
- P_C: 21 students (4% of the class)

The first noticeable pattern here is that the number of students attempting the bonus question decreased by approximately half for each successive practical assignment, which was counterintuitive to what was expected. The expectation was that students would become more confident and attempt more bonus questions as time progressed. A logical conclusion that can be drawn from this observation is that, since practical assignments become increasingly challenging, less and less students had the capacity or time (or even perhaps motivation) to attempt the bonus sections as the course progressed.

The use of modules as a bonus question was also included in the practical semester test, a three-hour long assessment testing on all the content covered during the semester. Only 37 students attempted the bonus question during the practical test. Out of the 35 students who sat for the sick test, none of them attempted the bonus question, which was unsurprising given that, *typically*, students who apply for deferred assessments at university tend to be those who did not prepare well for the main assessment opportunities. Since our practical semester test and the final examination have the same format and are on the same difficulty level, students can get promoted to the second semester without writing the exam if their semester mark is higher than 70%. When a student is promoted, their semester mark becomes their exam and final marks. It should be noted that students have the option to write the final examination even if they are promoted, but students rarely entertain this option.

Out of the 180 students who sat for the final examination (those who did not promote), only 11 students attempted the bonus question. Such a low number of students attempting the bonus question was not surprising since the highly performing students had already been promoted to the second semester and they did not sit for the exam. It should also be noted that most students who attempted the bonus question in the final exam did not perform well on the question.

3.2 Impact on Learning Outcomes

The previous section presented the results of having introduced C++20 modules as bonus content in the course. The results mainly focused on student participation in various assessments, including the final examination. The purpose of this section is to discuss the impact of having introduced modules on overall learning outcomes. It is difficult to quantitatively assess this impact since modules only formed part of the bonus section. However, the most important correlation worth mentioning is that the overwhelming majority (97%) of students who attempted the bonus content in one or more assessments throughout the semester eventually passed the course.

The most logical interpretation of this outcome is twofold: (1) students who had the capacity to attempt bonus questions were those who were already comfortable with mainstream concepts, which makes it unsurprising that they passed. A secondary observation made while marking the practical semester test was that (2) several students who were not particularly excellent in mainstream content were able to significantly boost their practical component mark by learning how to complete the bonus content (C++20 modules) and correctly executing what they had practiced.

A total of 123 (20.3%) students attempted C++20 modules in one or more instances during the semester and 119 of those students passed at the end of the semester. It is worth noting that the four students who failed all attempted bonus questions in only a single instance during the semester (none of the students attempted the bonus two or more times). For example, three of the four students only attempted the bonus when completing P_C (which may be an indication of desperate attempts to boost their practical mark towards the end of the semester) while only one attempted the bonus question in P_A. Figure 3 is a pareto chart showing that most attempts to use C++20 modules came from the students who eventually promoted from the course without having to write the final examination.

The key takeaway from Figure 3 is that most bonus attempts (using C++20 modules) during the semester came from the students who were eventually promoted from the course without having to sit for the final exam. Even more so, more than 95% of bonus attempts throughout the semester came from students who eventually succeeded in the course (either got promoted or passed after having written the final exam). This outcome was not surprising as students who excel at mainstream content often feel confident to engage with bonus concepts.

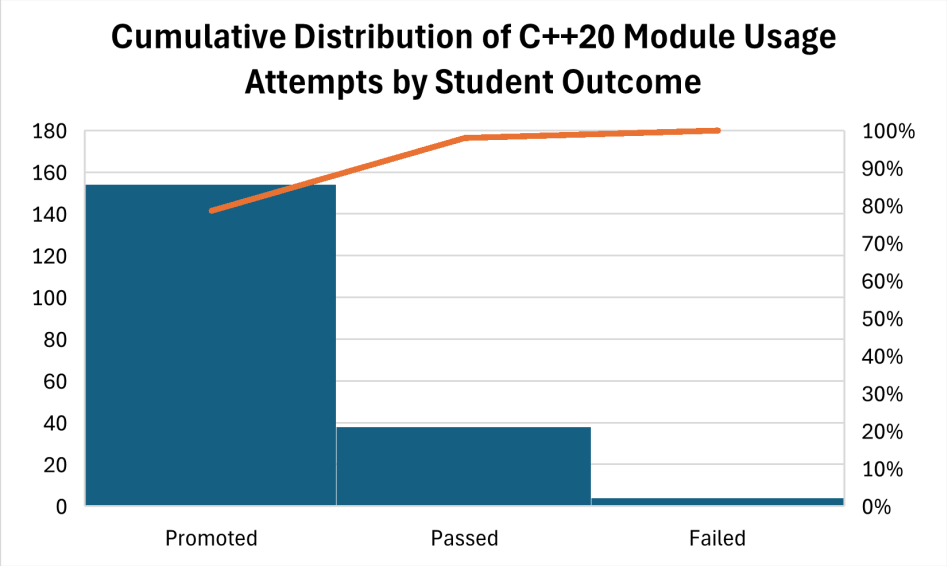

Figure 3: A Pareto analysis of C++20 module usage attempts across various assessments by student outcome.

The above results reflect student interest in engaging with C++20 modules during the first semester of the first-year computer science course. In summary, 123 out of 607 (20%) students engaged with C++20 modules in an official assessment and most of them were high-performing students. The following section reflects on the results as well as lessons learned.

4. Discussion

The introduction of C++20 modules was met with various interesting challenges, such as choosing the right compiler toolchain, creating conducive development and practice environments, and finding innovative ways to teach the bonus content without obstructing the delivery of mainstream content. Based on the level of engagement with C++20 modules in the first semester, modules have been integrated into the course as a mainstream topic in the second semester. This was done because we had verified that GCC can be used as a stable development platform with minimal setup overhead.

Introducing C++20 modules required leaning into constructivism, where students were encouraged to grapple with new concepts (including marking assistants who had to be taught how to mark the bonus sections). It became evident that following a scaffolding approach by first making the VPL the only platform for completing modular programming exercises resulted in a gentle introduction to the concepts. Once students were comfortable with the use of modules from a programming language perspective, they were then taught how to use the compiler to compile programs that make use of modules. This technology-enhanced approach to active learning resulted in a quantifiable interest in C++20 modules: students exhibited proactive learning behaviours and went on to seek additional cognitive challenge by repeatedly interacting with the bonus content.

The challenges and lessons learned from choosing to right compiler for our course as well as finding the right IDE resulted in the abandonment of Codeblocks in the second semester. The official programming platforms prescribed to students in the second semester were Notepad++ and the VPL. These two platforms were chosen specifically for the exposure they provide to the implementation details of C++ programs. Students who are not yet comfortable directly interacting with the compiler can use the VPL, which abstracts away the compilation process and allows students to focus only on coding. On the other hand, more advanced students who are willing to explore compilation from the ground up can use Notepad++ to write their programs and create batch files to compile the required executables. This approach is expected to provide the right level of engagement for both advanced and less experienced students.

5. Conclusion

This paper reported on a real-world case study where C++20 modules were introduced as a new and modern way of building programmer-defined libraries in a first-year computer science course. Modules were only introduced as part of the bonus topic of the course in the first semester, with the goal of making them

compulsory in the second semester (thus doing away with traditional header files). Based on the engagement with C++20 modules (20.3%) as part of the bonus content, the endeavour is viewed as a success because students are not forced to engage with bonus topics. Therefore, it is reasonable to say that 123 out of 607 students, who decided on their own to explore the topic, is a positive outcome.

In the second semester, when the topic becomes compulsory, 123 students will already have a degree of familiarity with it. It is also worth believing that the constructivist and project-based learning approach will further benefit the students who explored the topic in the first semester as they transition into the second semester due to the practical nature of the content delivery. When C++20 modules are explored theoretically in lectures, the students who already grappled with the practically are expected to have a more nuanced understanding of the concepts.

Future work should focus on addressing the technical challenges faced, such as choosing the best IDE for modern C++ and transitioning away from Codeblocks. Using Notepad++ as the official text editor can further reinforce students' understanding of the compilation process and use of batch files. This case study contributes to the wider educational discussion by giving an honest and detailed account – including quantitative results – on the struggles faced with introducing modern C++ topics to an undergraduate computer science class. The case study also gives light into the current capabilities of modern editors, such as Visual Studio Code to support C++ modular programming. The long-term goal is to continue gradually transitioning into modern C++ for this course as compilers continue to gradually support modern C++. This journey will lead to further exploration of the challenges and success that emerge.

Ethics Declaration

Ethical clearance was not required for this research.

AI Declaration

AI tools were not used to create this paper.

References

Chan, T., 1996. *Unix system programming using C++.* s.l.:Prentice-Hall, Inc..

Chun, W., 2001. *Core python programming.* s.l.:Prentice Hall Professional.

Felder, R. M. & Brent, R., 2009. Active learning: An introduction. *ASQ higher education brief,* Volume 2, p. 1–5.

Gibbons, P., 2002. *Scaffolding language, scaffolding learning.* s.l.:Heinemann Portsmouth, NH.

Holmes, J., 1994. *Building your own compiler with C++.* s.l.:Prentice Hall PTR.

Li, W., Sun, J. & Chen, H., 2019. Detecting Undefined Behaviors in CUDA C. *IEEE Access,* Volume 7, pp. 182559-182572.

Mack, K. & Ruud, R., 2019. *Unreal Engine 4 Virtual Reality Projects: Build Immersive, Real-world VR Applications Using UE4, C++, and Unreal Blueprints.* s.l.:Packt Publishing Ltd.

Reddy, M., 2011. *API Design for C++.* s.l.:Elsevier.

Sanders, J. & Kandrot, E., 2010. *CUDA by example: an introduction to general-purpose GPU programming.* s.l.:Addison-Wesley Professional.

Soto, M. S. & Figueroa, I., 2018. *Heuristic Evaluation of Code::Blocks as a Tool for First Year Programming Courses.* s.l., s.n., pp. 1-8.

Stroustrup, B., 1996. A history of C++ 1979–1991. In: *History of programming languages—II.* s.l.:s.n., p. 699–769.

Sutherland, B., 2014. *Learn C++ for game development.* s.l.:Apress.

Szalay, R. & Porkoláb, Z., 2025. Refactoring to Standard C++ 20 Modules. *Journal of Software: Evolution and Process,* Volume 37, p. e2736.

Tan, J., Guo, X., Zheng, W. & others, 2014. Case-based teaching using the Laboratory Animal System for learning C/C++ programming. *Computers & Education,* Volume 77, p. 39–49.

Von Glasersfeld, E., 2012. A constructivist approach to teaching. In: *Constructivism in education.* s.l.:Routledge, p. 3–15.