Eastwood: C Linting for Automated Code Style Enforcement in Programming Courses

Rowan Hart
Purdue University
West Lafayette, Indiana
hart111@purdue.edu

Brian Hays
Purdue University
West Lafayette, Indiana
bph412@gmail.com

Connor McMillin
Purdue University
West Lafayette, Indiana
mcmillinconnor@gmail.com

El Kindi Rezig
Massachusetts Institute of Technology
Cambridge, Massachusetts
elkindi@mit.edu

Gustavo Rodriguez-Rivera
Purdue University
West Lafayette, Indiana
grr@purdue.edu

Jeffrey A. Turkstra
Purdue University
West Lafayette, Indiana
jeff@purdue.edu

ABSTRACT

Computer Science students receive significant instruction towards writing functioning code that correctly satisfies requirements. Auto-graders have been shown effective at scalably running student code and determining whether the code correctly implements a given assignment or project. However, code functionality is only one component of "good" code, and there are few studies on the correlation between code style and code quality. There are even fewer studies contributing a tool equivalent to auto-graders for code style checking and grading. We put forth two contributions. First, a style guide for the C programming language focused on readability for student programs. Second, an automated linting tool *Eastwood* that provides on-demand style violation and fix feedback for students and automated style grading for course staff. Finally, we survey students and find a positive response to both a code standard and an automated tool to support the standard and make recommendations for the inclusion of both in programming focused courses based on these results.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **Applied computing → Computer-assisted instruction**.

KEYWORDS

code style, linting, c language, computer science education, automated assessment tools, automated feedback

ACM Reference Format:
Rowan Hart, Brian Hays, Connor McMillin, El Kindi Rezig, Gustavo Rodriguez-Rivera, and Jeffrey A. Turkstra. 2018. Eastwood: C Linting for Automated Code Style Enforcement in Programming Courses. In *SIGCSE 2023 Technical Symposium, March 2023, Toronto, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

SIGCSE TS 2023, March 2023, Toronto, Canada

https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Computer Science curriculum at the authors' university includes core classes which consist, with few exceptions, of courses primarily taught using the C programming language. These courses are generally large, up to several hundred students at a time, and make liberal use of auto-graders [1], including custom solutions as well as platforms such as Vocareum as class sizes increase, a trend that shows no signs of slowing [13]. Unlike automated grading for functionality of student code, professors implement style standards for student code as they see fit on a per-course basis, some with as little as a simple visual check by teaching assistants during grading. Some courses do not check student code for style at all. Some courses check student code during sessions with Teaching Assistants, but style is not a graded component of the course.

We posit instructing students to write code with proper style by adhering to programming best practice guidelines is a critical part of teaching programming. Ample anecdotal evidence is available from educators worldwide detailing questionable student choices when writing code, a fact often attributable to a lack of formal standard for those students to adhere to. Like learning syntax, language features, and algorithms, learning to program *well*, not simply to write programs that pass a set of tests, is critical to the development of quality programmers [14] and should be taught and evaluated at a commensurate level of rigor.

Producing students with the ability to write clean, safe code in addition to code that meets functional requirements is beneficial to students and the reputation of the institution. Industry and open source giants such as Google [5], LLVM [10], and Gnome [3] implement style and practice guidelines for programming and expect employees to adhere to them. It is prudent, then, to instruct and evaluate students in a similar manner to prepare them to make positive, usable contributions in their careers through meaningful, competent code. We present a comprehensive coding standard appropriate for new C programmers, as well as tooling for the scalable application and evaluation of this standard across even large classes.

2 RELATED WORKS

2.1 Auto-Graders

Prior work in the field of automated grading and course scaling is widely available, with automated systems and auto-graders filling sessions at *SIGCSE* frequently in the last decade. Commercially available grading systems such as Gradescope and Vocareum have pushed functionality assessment toward scalability. Ihantola et. al in 2010 created an overview of recent advancements and developments in automatic grading of Computer Science coursework [**?**]. This overview details advancements in learning management, including automatic testing of code functionality across languages and disciplines. However, this overview shows the distinct lack of tools specifically targeted at analyzing code style, only mentioning style once by way of Kirsti Ala-Mutka [1], who describes traditional linter tools such as Lint as well as advanced tools such as Checkstyle. Critically, Ala-Mutka connects programming style to "reliability, functionality, and maintainability" [1], a connection that appears to have degraded over time. Of auto-grading and automation papers put forth in SIGCSE in the past half decade, only a handful mention coding style, and none present a method for distributing the automated grading of coding style.

2.2 Compiler Messages

Compiler messaging is deeply related to coding style. Compilers, as the primary static analysis tool in use today, have become adept at providing messages related to uninitialized variables, improper boolean operations, and so forth. LLVM-based compilers such as clang, have made pioneering strides towards simplifying error messages to enable programmers to solve complex issues quickly using static analysis [8]. The GNU Compiler Collection has also begun intiatives towards improving messaging, implementing Fix-It hints that point to specific code locations and not only display an error, warning, or note but also provide a suggested fix [**?**]. Some fixes provided by these systems can even be applied automatically, reducing programmer burden significantly and streamlining debugging. Research has been conducted into the effectiveness of improved, human readable compiler errors with mixed [12] to positive results [2] measured in increased student performance by Pettit et. al and Becker, respectively. Becker points out that "students often have little experience to draw on, leaving compiler error messages as their primary guidance on error correction". This is doubly true in large courses where office hours may be crowded, reducing the availability of personalized assistance.

2.3 Style Analysis

Despite the small quantity of recent code-style focused research, effort has been put forth in analyzing code style, particularly in languages higher level than C. Moghadam et. al put forth a system for analyzing code style, however in their work they "refer not to mechanical coding conventions (indentation, punctuation, naming of variables, and so on), but to the effective use of programming idioms" [11]. This concept is not entirely unrelated to conventional style, as idiomatic programming in C in particular is linked to convention, especially with regards to macro and pointer use. Leite and Blanco present a comparison of human feedback as compared to automatic feedback [7], however their automated system does not deliver syntactic feedback, only functionality-related feedback while their human feedback included syntactic feedback. They uncover a marginal improvement in the coding style of students who received style feedback. This suggests that as expected, feedback on style leads to improved style. This is an important observation, if not a novel one, and is an observation that we seek to directly leverage by making the style feedback provided by humans in Leite and Blanco's work possible via automated checks, which eliminates or greatly reduces the need to commit teaching resources to style assistance.

Outside of educational environments, programming style receives middling attention. As previously mentioned, corporations such as Google [5] and large open source projects commonly provide and strictly enforce style guides. Google also maintains a tool called `cpplint` [6] that automatically checks code for conformance to their style guide. This tool is a precursor to the set of checks Google provides for Clang-Tidy, and is in fact very similar to the initial iteration of `Eastwood` in implementing a custom parser. Also similarly to `Eastwood`, cpplint does not automatically fix code, it simply outputs errors. The motivation for this decision with respect to `Eastwood` is covered in greater depth in the Methodology section.

Yang et. al provide a method for using static Abstract Syntax Tree (AST) based methods for analyzing programming style using a rule set. They focus on Java, however the methodology is exceedingly similar to that which we have put forth with *Eastwood*, using AST traversal to analyze code in the same way a compiler would. In addition, Yang et. al provide studies of running their analyzer on large open source projects with striking results, uncovering hundreds of style violations. They recommend future work into developing analyzers for source code and instituting code style training for developers, a goal we propose a novel solution for here.

2.4 Linters

There are myriad existing tools designed to automatically format code, beginning from the Unix `indent` utility and evolving into to modern solutions such as Clang-Tidy checks that automatically fix style violations as they check code.

3 HISTORY

The computer science department at our university has always had some form or another of code style requirements and analysis, as mentioned previously. However, for the C language specifically, a fully-functional automated tool and standard did not exist until 2019, when the initial version of the *Eastwood* tool was developed and implemented by two teaching assistants for our university's course "Programming in C", which all Computer Science students are required to pass to complete a bachelors degree.

[TODO: statement from Jeff on why an automated tool and proper code standard was needed]

[TODO: statement from Brian and Connor (in progress) on the history of the development of Eastwood 1.0]

Despite the success of the initial version of *Eastwood*, students and teaching assistants observed several critical issues with the platform. Because it was built atop top of a Flex and Bison based parser for the ANSI C11 standard, the tool was unable to run on many external libraries, especially libraries employing GNU extensions to the C language such as GTK-3.0 and Cairo, both of which are used in projects in introductory C courses and in later courses employing the C language at our university. In addition, the structure of *Eastwood* as a parser with no attached type or preprocessor system meant it experienced intermittent failures to resolve types and resorted to imperfect methods of analyzing code. This was mostly confined to edge cases, in most cases having little trouble analyzing code submitted by students, however student feedback and staff experience indicated that a stronger tool would be appreciated. Rather than add the necessary functionality to the current system, we explored alternative platforms with a larger feature set we could build atop to accomplish our goals in a more extensible and maintainable fashion.

After considering several options, we elected to re-implement our current C Code Standard by converting the rules to Clang-Tidy [9] checks. The wealth of features provided by the LLVM Project via Clang-Tidy facilitated re-implementation by a single developer and allowed creating a more flexible tool that could feasibly be modified and extended to accomplish the goals of not only C programming courses, but C++ and Objective-C programming courses. In addition, the LLVM project provides interfaces to deeply analyze a program's AST, a feature that can be leveraged to combine style analysis with functionality checks that provide a more holistic view of a student's grasp of the language and solution to a problem.

4 METHODOLOGY

Eastwood is based on a code standard developed and used in C language courses, particularly "Programming in C" and "Systems Programming". [TODO: insert statements from Jeff about the development and contribution of the code standard here. I don't know much about its origin/inspiration/anything it was based on].

For each section and subsection, we implemented a Clang-Tidy check that verifies whether a student's code meets that part of the standard. If a violation is detected, *Eastwood* gives a warning of the exact violation, the location it occurs, and in most cases suggests a change to fix the violation. The output includes an arrow indicating the exact source location or source range range containing the error (Listing 1). Automatic fixes are not currently enabled, but are supported by the Clang-Tidy framework and intentionally disabled for our use case. We disable automatic fixes to bring the students into the code formatting and style process – in typical industry or academic software development a format would be applied automatically, but we believe causing the students to think about their actions and style is helpful to develop good habits. However, any educator wishing to adopt our framework could enable automated fixing to allow their students to apply fixes and conform to the course coding standard mostly automatically. Each of these checks is implemented in a single C++ source file and corresponding header, and are implemented in a total of 7000 LOC. Despite the non-trivial nature of learning the LLVM and Clang-Tidy framework in order to implement style checks, this is a comparatively small undertaking as compared to creating a full C language preprocessor, parser, and static analysis system and can be feasibly extended or modified by other educators seeking to implement their own standard.

The default checks can be divided into two categories: syntactic checks and usage checks. Syntactic checks involve purely appearance-related properties of code such as whether binary operators are surrounded by spaces, whether functions have descriptive header comments, or line and function length. Usage checks are a level deeper, determining factors such as whether all variables are initialized at the time they are declared, checking all null values are appropriately typed, or determining whether an expression contains multiple assignment sub-expressions. Generally, syntactic checks utilize the concept of *Matchers* to locate specific nodes in the program AST to perform checks: for example the check to ensure all binary operators have a space preceeding and following the operator matches all binary operators, then uses the clang lexer to obtain tokens around the operator. Usage checks leverage *Matchers* as well, but tend not to use the clang lexer directly, preferring to utilize structural checks and AST visitors to check assertions about the code. For example: every variable declaration is checked to ensure that the declaration is also a definition to enforce the idea that "Resource Allocation Is Initialization".

These default checks necessarily closely reflect the specific code standard in use for these C programming courses, however many are easily adaptable to a slightly different standard, and the framework for creating these rules serves as a model for how such a tool can be created using modern techniques, languages, and frameworks.

These checks primarily leverage the Lexer and Matcher functionality of the LLVM libraries to locate instances of problem code and report them to the student. This has become a critical design decision. Rather than utilize the code standard and *Eastwood* linter solely to grade students, all students are provided access to the linter throughout the semester as a binary and trained on its use for self-evaluation. The objective is twofold. First, our position is that students will develop good coding habits not by returning to their code after writing it and fixing code standard mistakes, but by continuously writing code, evaluating coding style, testing functionality, and repeating the cycle. Second, the intent of the code standard is not to create a roadblock of tedious work for students but to provide a format that encourages clean, readable, functional code. By providing a tool to automatically check much of the standard, we aim to make adherence to this standard less tedious and

more habitual.

```
every_error.c:1:1: warning: Corresponding header file include must be first and
have same basename as source file. [eastwood-Rule8aCheck]
#include <other.h>
^
every_error.c:1:10: warning: Local includes must be included with "...", not
angle braces. [eastwood-Rule8gCheck]
note: this fix will not be applied because it overlaps with another fix
every_error.c:3:5: warning: Variable declaration without definition is
forbidden. [eastwood-Rule12bCheck]
int globalval;
    ^
every_error.c:3:5: warning: Global variable 'globalval' doesn't conform to
global naming scheme. [eastwood-Rule1dCheck]
int globalval;
    ^~~~~~~~~
    g_globalval
every_error.c:5:9: warning: 'H' is not all uppercase, separated by underscores,
and >= 2 characters in length. [eastwood-Rule1cCheck]
#define H 1
        ^
        H
every_error.c:5:11: warning: 'H' initializer is non-string constant and not
surrounded by parentheses. [eastwood-Rule1cCheck]
#define H 1
          ^
          (1)
every_error.c:7:1: warning: All preprocessor directives outside of a function
should be placed at the top of the file after include directives.
[eastwood-Rule3dCheck]
#define goodbye ("goodbye")
^
every_error.c:7:9: warning: 'goodbye' is not all uppercase, separated by
underscores, and >= 2 characters in length. [eastwood-Rule1cCheck]
#define goodbye ("goodbye")
        ^~~~~~
        GOODBYE
every_error.c:7:17: warning: 'goodbye' initializer is string constant and
surrounded by parentheses. [eastwood-Rule1cCheck]
#define goodbye ("goodbye")
                ^~~~~~~~~~
                "goodbye"
every_error.c:9:1: warning: Malformed function header comment for function
foo. [eastwood-Rule7aCheck]
/* foo */
^
every_error.c:11:8: warning: No space permitted between function name and
parameter list. [eastwood-Rule3fCheck]
int foo () {
       ^~
every_error.c:12:5: warning: Incorrect indentation level. Expected 2, got 4
[eastwood-Rule4aCheck]
    int looooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong;
  ~~^~~~
every_error.c:12:9: warning: Variable declaration without definition is
forbidden. [eastwood-Rule12bCheck]
    int looooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong;
        ^
every_error.c:12:82: warning: Line length must be less than 80 characters
[eastwood-Rule2aCheck]
    int looooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong;
                                                                                 ~~^
every_error.c:14:13: warning: There must be exactly one space between
parenthesis and open brace. [eastwood-Rule3aCheck]
  if (a == 5){
            ^~
every_error.c:15:6: warning: Leading space required. [eastwood-Rule3bCheck]
    a+=3;
     ^
every_error.c:15:8: warning: Trailing space required. [eastwood-Rule3bCheck]
    a+=3;
       ^
every_error.c:17:18: warning: Single space required after ';'. [eastwood-Rule3cCheck]
  for (int i = 0;i<a;i++) {
                  ^
every_error.c:17:19: warning: Leading space required. [eastwood-Rule3bCheck]
  for (int i = 0;i<a;i++) {
                   ^
every_error.c:17:20: warning: Trailing space required. [eastwood-Rule3bCheck]
  for (int i = 0;i<a;i++) {
                    ^
every_error.c:17:22: warning: Single space required after ';'. [eastwood-Rule3cCheck]
  for (int i = 0;i<a;i++) {
                      ^
every_error.c:17:28: warning: Trailing whitespace. [eastwood-Rule3eCheck]
  for (int i = 0;i<a;i++) {
                           ^
every_error.c:22:3: warning: Use of goto. [eastwood-Rule11eCheck]
  goto end;
  ^~~~~~~
every_error.c:34:8: warning: Line-broken parameter is not aligned with first
parameter. Parameters should be aligned in column 9 (got 8). [eastwood-Rule4bCheck]
       char *b) {
       ^
every_error.c:37:4: warning: While statement in a do-while must be on same
line as the closing brace. [eastwood-Rule4cCheck]
  }
   ^
every_error.c:38:27: warning: Comments must appear above code except for else,
case, #defines, or declarations [eastwood-Rule5bCheck]
  while (*(a++) != '\0'); // Bad comment
                          ^~~~~~~~~~~~~
```

Listing 1: Example of *Eastwood* output, formatted for width

[TODO: insert some details about how code standard grading works and how the linter is integrated into that process. This is best written by Jeff].

5 METRICS AND RESULTS

To gauge student sentiment concerning the helpfulness of both the code standard and linter, students were asked to voluntarily complete two anonymous surveys, one at the halfway point and one at the end of the Spring 2021 semester of "Programming in C", the first course at our University in which students learn to use the C language. As a foundational course, enrollment is high at approximately 200 students. For first six weeks of the course, students used the "old" version of *Eastwood* implemented with a custom parser. For the remainder of the course, students used the "new" version of *Eastwood* implemented using Clang-Tidy once development of the new version reached an appropriate stage for widespread use. The surveys collected responses from students about both the code standard used in the course and *Eastwood*, the linting tool to automatically check adherence to the standard. Specifically, the survey sought to determine students takeaways about two factors. First, whether they felt that using the code standard and linting tool caused them to write better, more readable code with less bugs. Second, whether they had a positive reaction to the requirement to follow a coding standard and intended to continue to follow some personal standard in the future. For tool development reasons, the end of semester survey also asked about the differece between the "new" Clang-Tidy based linter and the "old" custom parser based linter. When only one value is discussed for a survey response, the value will be from the end of semester survey concerning the current "new" linter, as that linter is the primary contribution of this work.

As demonstrated by the results of the student survey regarding the linting tool and code standard, student opinion is generally very positive. This indicates that despite the fact that requiring adherence to a standard could be perceived as "busy work" or not a direct contribution to writing code that passes test cases and earns a good score for functionality, students understand the value writing well structured and formatted code represents.

More than 80% of students reported agreement that *Eastwood* makes it easier to follow the meet the existing code standard. This means the tool accomplishes its primary goal, as the code standard has always been a component of the course grade. Likewise, over 75% of students reported that the linter saves time meeting the

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Question
25.9%	57.1%	11.6%	4.1%	1.4%	The new linter makes it easier to meet the code standard.
33.3%	42.9%	15.6%	6.1%	2%	The new linter saves time meeting the code standard.
23.1%	54.4%	15.6%	4.8%	2%	The new linter effectively checks whether my code meets the standard.
33.3%	50.3%	12.2%	2.7%	2.7%	The new linter helps me accurately locate code standard violations in my code.
19.7%	47.6%	28.6%	3.4%	0.7%	The new linter helps improve the quality of my code.
8.3%	31%	32.4%	22.8%	5.5%	The new linter helps me find bugs in my code.
23.1%	50.3%	17.7%	6.8%	2%	The new linter helps me make my code more readable.
15.8%	48.6%	19.2%	11.6%	4.8%	Following the code standard improves my code quality.
14.4%	38.4%	24%	16.6%	6.8%	I will continue to use the...Code Standard in future C programming...
21.9%	49.3%	13.7%	5.5%	9.6%	I will continue to use some code standard in future C programming...

Table 1: Results of Student Survey

code standard. This means the linter has certainly accomplished a secondary goal of reducing the tedium required to help their code conform to the standard. Anecdotal evidence from observing students during lab meetings suggests that because the linter is able to quickly check code, students use it continually while they work instead of waiting until the end to fix all formatting errors. We find this very meaningful, as it suggests students are integrating style and formatting into their workflow, a habit we have designed the code standard and linting tool to facilitate.

The next two questions, whether the linter "effectively checks whether my code meets the standard" (77.5% agreement) and whether the linter "helps me accurately locate code standard violations..." (83.6% agreement) sought to understand how **well** *Eastwood* was able to perform its checks. Once again, the highly positive responses suggest that the tool works accurately and correctly in helping students meet the set standard.

The next three questions were the least specific, but attempted to discern whether students felt that *Eastwood* helped students write "better" code by asking about three distinct areas. First, we asked about "quality" without giving a definition of "quality", thereby leaving students to answer subjectively. 67.3% of students agreed or strongly agreed that *Eastwood* helped them write higher quality code, a result we feel is very indicative of how simple guidelines and tools to help students painlessly follow them can improve habits and results. Unsurprisingly, less than 40% of students agreed that *Eastwood* helped them find bugs in their code. Finding bugs is not a goal of *Eastwood* and in fact there are zero rules built into the tool that detect the definite presence of a programming error. That even 40% of students agreed with this statement then, is extremely interesting. We suspect that by writing better formatted and well-styled code, students were more easily able to *find bugs themselves*, a deeper and even more encouraging result than anticipated. This is difficult to quantitatively ascertain, but from anecdotal discussions with students we believe this is the case. Finally, nearly 75% of students felt *Eastwood* helped make their code more readable.

Finally, we asked questions regarding the code standard itself, instead of about the *Eastwood* linter. Anecdotally, students find some of the points in the code standard enforced in our course unnecessary or reflective of older K&R C programming guidelines. Thus, it was unsurprising that only 52% of students agreed that they would use the code standard from this course in furure C programming. The over 70% of students who agree that they will continue to use some code standard, not necessarily the standard used in this particular course, in the future is a strong positive result. This demonstrates that overall, students have seen the value of having some standard to hold themselves to and more importantly, intend to continue to do so of their own volition.

Overall, the results of this survey suggest to us that *Eastwood* is effective in meeting its primary goal of helping students more easily meet the course code standard we have set. We also observed a positive indication that students both understand and value the role of code style, consistency, and formatting in the software development process.

6 CONCLUSIONS AND FUTURE WORK

We propose two ideas concerning assessment of student code projects in Computer Science. First, we recommend that programming courses adopt a code style and best practices standard for code formatting, organization, and syntax in addition to language-specific best practices. We base this recommendation on previously referenced statements by Ala-Mutka and Yang, both of whom present information suggesting a tangible association between code style and overall code quality. Second, we recommend that in adopting a code style guide, automated tooling such as the *Eastwood* linter be utilized to the fullest possible extent to provide the best experience for staff and students. Courses typically have only a few Graduate Teaching Assistants and several Undergraduate Teaching Assistants. As put forth by Dickson, Dragon, and Lee [4] it is feasible to delegate grading of code standard adherence to Undergraduates,

however doing so is not an effective use of time, especially when automated grading systems are available and possible.

We put forth *Eastwood*, one such example of an end to end solution that provides instant, feedback and suggestions to students. Such a system helps encourage good programming habits that will ideally stay with students as they continue in their careers. It also reduces the workload on students and course staff in the software development and grading processes, respectively.

There is ample future work in this area. Defining code standards is a somewhat subjective matter, and implementing automated checks for those standardized requirements will provide a continuous engineering challenge. Improved static analysis techniques would allow a code standard to check for functionality errors that may not be observed by simple output-based automated test systems such as Vocareum employed heavily by university programs. In summation: code style is an exciting opportunity to reinforce student skills that are at present largely left by the wayside.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102. https://doi.org/10.1080/08993400500150747 arXiv:https://doi.org/10.1080/08993400500150747

[2] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[3] Gnome Developer. 2020. *C Coding Style*. https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en

[4] Paul E. Dickson, Toby Dragon, and Adam Lee. 2017. Using Undergraduate Teaching Assistants in Small Classes. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 165–170. https://doi.org/10.1145/3017680.3017725

[5] Google. 2020. *Google C++ Style Guide*. https://google.github.io/styleguide/cppguide.html

[6] Google. 2021. *cpplint*. https://github.com/google/styleguide

[7] Abe Leite and Saúl A. Blanco. 2020. Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 44–50. https://doi.org/10.1145/3328778.3366921

[8] LLVM. 2020. *Available Checkers*. https://clang-analyzer.llvm.org/available_checks.html

[9] LLVM. 2020. *Clang-Tidy*. https://clang.llvm.org/extra/clang-tidy/

[10] LLVM. 2020. *LLVM Coding Standards*. https://llvm.org/docs/CodingStandards.html

[11] Joseph Bahman Moghadam, Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. 2015. AutoStyle: Toward Coding Style Feedback at Scale. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale* (Vancouver, BC, Canada) *(L@S '15)*. Association for Computing Machinery, New York, NY, USA, 261–266. https://doi.org/10.1145/2724660.2728672

[12] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students? Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 465–470. https://doi.org/10.1145/3017680.3017768

[13] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616

[14] Chunyu Yang, Yan Liu, and Jia Yu. 2018. Exploring Violations of Programming Styles: Insights from Open Source Projects. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence* (Shenzhen, China) *(CSAI '18)*. Association for Computing Machinery, New York, NY, USA, 185–189. https://doi.org/10.1145/3297156.3297227