Eastwood-Tidy: C Linting for Automated Code Style Assessment 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. Autograders 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-Tidy` providing 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. 2023. `Eastwood-Tidy`: C Linting for Automated Code Style Assessment in Programming Courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569817

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569817

1 INTRODUCTION

The C programming language is the second most popular programming language according to the TIOBE popularity index [25]. C is primarily used in operating systems, embedded systems, and performance-critical applications. Due to this and the language's simplicity, it is often taught in foundational or introductory programming courses at major universities. These courses can often be large, with enrollments approaching one thousand students. The scale of these courses is almost always enabled by using auto-graders [1], both custom solutions and commercial platforms such as Vocareum [27]. As class sizes further increase—a trend that shows no signs of slowing [29]—these auto-graders become continually more helpful, if not necessary. In addition to grading for functionality, courses often implement style standards for code. `Eastwood-Tidy` and the included code standard provides one proposed and tested approach to address this additional, and often overlooked, challenge.

Ensuring that students write code that adheres to some consistent style is a critical part of teaching students to create quality code. The computer science field is rife with stories involving "spaghetti code" and other descriptive terms for difficult to read and comprehend code. Such poor quality code is often attributable to a lack of formal education and experience as well as a lack of exposure to code style standards. Similar to learning syntax, language features, and algorithms, learning to write code *well* is critical to the development of good programmers [30] and should be emphasized and evaluated at a commensurate level of rigor.

Fostering students' ability to write clean, safe code in addition to code that meets functional requirements prepares them for professional team-oriented programming. Industry and open source giants such as Google [11], LLVM [19], and Gnome [6] implement style guides for programming and expect employees or contributors to adhere to them. It is prudent to instruct and evaluate students similarly to prepare them to make positive, usable contributions in their careers through readable and functioning code. `Eastwood-Tidy`, and the associated coding standard, provide one approach appropriate for new C programmers. The tooling specifically aids in evaluating the presented standard across large classes.

2 RELATED WORKS

2.1 Auto-Graders

Prior work in 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. In [14], Ihantola et. al. 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 Ala-Mutka [1], who describes traditional linting tools such as Lint and advanced tools such as Checkstyle. Critically, Ala-Mutka connects programming style to “reliability, functionality, and maintainability” [1]. Of auto-grading and automation papers put forth in SIGCSE in the past half decade, only a handful ([4], [13], [28], [5], and [15]) mention coding style and only *Hyperstyle* [4] presents a method for scalable style analysis, but focuses on using existing industry tools with pre-configured defaults instead of custom style requirements on a per-course basis.

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, etc. LLVM-based compilers such as clang [20] have made pioneering strides toward simplifying error messages to enable programmers to solve complex issues quickly using static analysis [17]. The GNU Compiler Collection [9] has also begun initiatives towards improving messaging—implementing Fix-It hints [10] that point to specific code locations and provide suggested fixes in addition to the error, warning, or note. Some fixes these systems provide 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 [24] to positive results [3] 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, an effort has been put into analyzing code style. Particularly with higher-level languages [22]. 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” [23]. This concept is not entirely unrelated to conventional style, as idiomatic programming in C, particularly, is linked to a convention, especially regarding to macro and pointer use. Leite and Blanco present a comparison of human feedback as compared to automatic feedback [16], however their automated system does not deliver syntactic feedback, only functionality-related feedback, while their human feedback included syntax-related comments. They uncovered marginal improvement in the coding style of students who received style feedback. A more comprehensive, on-demand method of delivering this feedback style may improve results over manual human delivery. This is an important observation, if not a novel one, and automation of style feedback similar to feedback provided by humans in Leite and Blanco’s work eliminates or dramatically reduces the need to commit teaching resources to style assistance. Automating this feedback also ensures students always have up-to-date information on the current state of their code as they work.

Outside of educational environments, programming style receives significant attention. As previously mentioned, corporations such as Google [11] and large open source projects commonly provide and strictly enforce style guides. Google also maintains a tool called `cpplint` [12] 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 `Eastwood` in implementing a custom parser. Also similarly to `Eastwood` and `Eastwood-Tidy`, `cpplint` does not automatically fix code; it simply outputs errors. The motivation for this decision for `Eastwood-Tidy` is covered in greater depth in the Section 4.

Yang et al. [30] 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 the AST traversals `Eastwood-Tidy` uses to analyze code. 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, future work which `Eastwood-Tidy` addresses. In particular, the AST of programs is leveraged to disallow multiple assignments, require separating complex logical expressions, and check indentation of multi-line expressions.

2.4 Formatters

Numerous existing tools are designed to automatically format code, beginning from the Unix `indent` utility and evolving into modern solutions such as Clang-Tidy checks that automatically fix style and functionality violations as they check code. `Clang-Format` [21] is widely used for the C language, and is even integrated into most Integrated Development Environments such as Visual Studio Code. Most popular languages provide an automatic formatter, with varying degrees of configurability. Some, such as the famous `Black` [7] formatter for Python, provide almost no user configurability by design. Using automated formatting instead of an error reporting system may increase student reliance on automated formatting too early in their curriculum. Baniassad et al. [2] studied this effect with respect to auto-graders, with inconclusive results. Removing student access to the grader increases student engagement in testing at the cost of stress, a tradeoff that makes less sense for style testing than for functionality testing. Instead, guiding students toward careful consideration and manual formatting of their code while providing feedback when they have successfully done so requires students to pay manual attention to their style habits without causing panic.

3 HISTORY

Eastwood-Tidy and its associated code standard (see Standard I) is the result of several iterations of development to support a curriculum-required C programming course. Instructors of the course introduced this code standard and a grading policy change to add a small (15%) weight per assignment for code style. It has two benefits. First, students have an opportunity for every assignment to guarantee some points simply by writing properly formatted code following best practices. Second, a standard helps students form good programming habits that benefit them far beyond a single course. The first iteration of Eastwood was written in BASH and used complex regular expressions to check a small subset of the code standard. This script worked well on correctly formatted code, but could not quickly diagnose issues with incorrectly formatted code, which comprises the majority of code requiring accurate error reporting. This shortcoming is primarily due to the inherent inability of regular expressions to process context, precisely the context of a C program.

A second iteration, called Eastwood, was built using Bison and Flex. Eastwood used an extended C99 grammar to parse code submissions as code rather than text. An actual parser enabled checking variable naming conventions, forbidden statements, a grouping of logical expressions, and more. Enhanced error reporting allowed students to use Eastwood on-demand while completing assignments, eased conforming to the code standard, and enabled a faster grading process. Despite the success of Eastwood, students and teaching assistants observed several critical issues with the platform. Because it was built using a Flex and Bison-based parser for the ANSI C11 standard, the tool could not run on many external libraries. Especially libraries employing GNU extensions to the C language, such as GTK and Cairo, are used in projects in introductory C courses and later courses employing the C language at the authors' institution. 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 heuristics. Rather than add the necessary functionality to the current system, developers attempted to identify platforms with more extensive feature sets that would meet the requirements.

After considering several options, leveraging the LLVM framework became the clear path forward. Each code standard rule was converted into a Clang-Tidy [18] check, and the previously tedious tasks of lexing, parsing and AST analysis were nearly entirely handed off to existing LLVM functionality. The wealth of features provided by the LLVM Project via Clang-Tidy facilitated re-implementation by a single developer and allowed the creation of 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 code provided interfaces to analyze a program's AST deeply. This feature was leveraged to combine style analysis with functionality checks to provide a more holistic evaluation of a student's grasp of the language and solution to a problem. For example, neither Eastwood nor the original BASH script could check whether a FILE pointer is fclose'd after use, but this deep check is possible using LLVM.

4 SYSTEM DESIGN

Eastwood-Tidy assesses a coding standard that is used in programming courses at our university utilizing the C language—particularly the Introductory C Programming and Systems Programming courses. The code standard was inspired by the Google C++ Style Guide [12] with more concrete requirements around spacing and indentation. The code standard draws heavily from the Linux Kernel code standard [26], for C language-specific rules, especially for comment formatting, function size, naming, and spacing. It also adds several defensive programming requirements that should become habitual, such as NULL-ing free'd pointers.

The tool is run from the command line by issuing a simple `linter ./myfile.c` command either on a University machine (where Eastwood-Tidy is provided in the system PATH by default, or using a downloaded binary. For each section and subsection, an instructional assistant implemented a Clang-Tidy check that verifies whether a student's code meets a particular part of the standard. If a violation is detected, Eastwood-Tidy warns of the type of violation, the line and column location it occurs, and in many cases, suggests a change to fix the violation. The output includes an arrow indicating the exact source location or source range containing the error. Listing 1 shows an example of C source with the associated output (edited for brevity). Each of these checks is implemented in a single C++ source file with corresponding header file and are implemented in approximately seven thousand lines of C++ total. The check for rule 6.A for example, that ensures each set of logical comparisons is surrounded by parentheses, consists of a 95 line C++ source and 45 line header. Indentation checks are the largest, around 700 lines. Despite the complexity of LLVM and Clang-Tidy frameworks, this is a comparatively small undertaking as compared to creating a full C language preprocessor, parser, and static analysis system. The software is also architected to allow other educators to modify and extend the system to implement their standards. For example, each check Eastwood-Tidy implements can be toggled individually. Many parameters such as indentation amount (which is set to 2 spaces by default) can be modified by changing definitions when building without any additional code.

Automatic fixes are also supported. The Clang-Tidy framework can apply fixes based on suggestions implemented in Eastwood-Tidy for a majority of code standard violations. These automatic fixes are disabled by default to bring the students into the code formatting and style process. A format would be applied automatically in a specific industry or academic software development setting.

Requiring students to fix their errors manually helps them internalize a standard while creating more straightforward code for themselves and instructional staff to read, understand, and debug. However, any educator wishing to adopt Eastwood-Tidy could enable automated fixes, allowing their students to apply fixes and conform to the course coding standard mostly automatically.

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

```c
#include "x.h"
#include <other.h>
#define H 1
#define T ("bad")
int badglobal;

int add_values (int first_value,
       int second_value) {
  first_value = second_value + first_value;
    return first_value;
} // add values

int main() {
  int value = add_values(1, 2);
  if (value == 1) {
    return 1447;
  }
  value++;
  return value;
}
```

```
x.c:3:11: warning: 'H' initializer is non-string constant and not surrounded by
↪  parentheses. [eastwood-Rule1cCheck]
#define H 1
          ^
          (1)
x.c:5:5: warning: Variable declaration without definition is forbidden.
↪  [eastwood-Rule12bCheck]
int badglobal;
    ^
x.c:5:5: warning: Global variable 'badglobal' doesn't conform to global naming scheme.
↪  [eastwood-Rule1dCheck]
int badglobal;
    ^~~~~~~~~
    g_badglobal
x.c:7:1: warning: Missing header comment for function add_values.
↪  [eastwood-Rule7aCheck]
int add_values (int first_value,
^
x.c:7:15: warning: No space permitted between function name and parameter list.
↪  [eastwood-Rule3fCheck]
int add_values (int first_value,
              ^~
x.c:11:3: warning: End of function comment is malformed. Got "// add values" Expected
↪  "/* add_values() */". [eastwood-Rule5dCheck]
} // add values
  ^~~~~~~~~~~~~
  /* add_values() */
x.c:16:12: note: embedded constant of type 'integerLiteral'. [eastwood-Rule11dCheck]
    return 1447;
           ^
```

Listing 1: Example of a C program and a partial snippet of output generated by `Eastwood-Tidy`

level deeper, determining factors such as whether all variables are initialized when 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 ensuring all binary operators have a space preceding and following the operator first matches all binary operators, then uses the clang lexer to obtain tokens around the operator. Usage checks also leverage *Matchers* but tend not to use the clang lexer directly, preferring to utilize structural checks and AST visitors to check assertions about the code. E.g., every variable declaration is checked to ensure that the declaration is also a definition to enforce the critical idea "Resource Allocation Is Initialization."

These default checks 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, which has become a critical design decision. `Eastwood-Tidy` is used to grade students' submissions. It is also provided to students to self-check their code while completing programming assignments. The objective is twofold. First, to ensure students develop good coding habits, not by returning to their code after writing it and fixing code standard mistakes, but by continuously writing quality code from the beginning. Second, to provide a format encouraging clean, readable, and functional code. By providing a tool to check most of the standard automatically, adherence to this standard becomes less tedious and more habitual.

5 METRICS AND RESULTS

Students were asked to voluntarily complete two anonymous surveys (see Table 1) to gauge student sentiment concerning the helpfulness of the code standard and `Eastwood-Tidy`. These surveys were conducted at the beginning and end of the Spring 2021 offering of an introductory C programming course. As a first-year course, enrollment is high at nearly 500 students. The surveys collected students' responses about the code standard used in the course and `Eastwood-Tidy`. Specifically, the survey sought to determine students' takeaways about two factors. First, whether they felt using the code standard and linting tool caused them to write better, more readable code with fewer bugs. Second, whether they positively reacted 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 difference between `Eastwood-Tidy` and `Eastwood`. There were 186 responses to the survey.

As demonstrated by the results (see Table 1) 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 earns a good score for functionality, students understand the value in writing well-structured and formatted code.

More than 80% of students reported agreement that `Eastwood-Tidy` makes it easier to adhere to the existing code standard. This means the tool accomplishes its primary goal, as the code standard has

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

always been a component of the course grade. Likewise, over 75% of students reported that Eastwood-Tidy saves time meeting the code standard. This data suggests Eastwood-Tidy 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 suggest that because Eastwood-Tidy can quickly check code, students use it continually while they work instead of waiting until the end to fix all formatting errors. This result is meaningful as it suggests students are integrating style and formatting into their workflow, a habit the code standard and linting tools were originally designed to facilitate.

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

The following three questions were the least specific but attempted to discern whether Eastwood-Tidy helped students write "better" code by asking about three distinct areas. First, students were asked about "quality" without defining "quality", thereby leaving students to answer subjectively. 67.3% of students agreed or strongly agreed that Eastwood-Tidy helped them write higher-quality code. This result indicates how simple guidelines and tools can help students improve habits and results. Unsurprisingly, less than 40% of students agreed that Eastwood-Tidy helped them find bugs in their code. Finding bugs is not a goal of Eastwood-Tidy, and there are zero rules built into the tool that detects the definite presence of a programming error. That even 40% of students agreed with this statement is interesting. One explanation is that by writing a better formatted and well-styled code, students were more easily able to *find bugs themselves*, an even more encouraging result than anticipated. Quantitatively, it isn't easy to ascertain, but discussions with students suggest this is the case. Finally, nearly 75% of students felt Eastwood-Tidy helped make their code more readable.

Students were also given questions regarding the code standard itself instead of the Eastwood-Tidy linter. Students have reported finding some of the points in the code standard 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 future C programming. Over 70% of students agree that they will continue to use some coding standard in the future. *Any* code standard assessment at scale in programming courses can have a positive impact. Despite using the code standard in Standard I, the authors do not necessarily prefer it over any other well-defined and robust standard. The goal to form a habit of following some standard is effective, according to these data, which is an encouraging result in itself. Overall, students appear to recognize the value of having some standard to hold themselves to and, perhaps more importantly, intend to continue to do so of their own volition.

The survey results suggest that Eastwood-Tidy effectively meets its primary goal of helping students more easily meet the included course code standard. Students indicate via their responses both an understanding and appreciation for the role of code style, consistency, and formatting in the software development process. Equally important, most students intend to continue utilizing a coding standard for themselves in the future.

Several authors were UTAs in C courses prior to implementing Eastwood-Tidy and spent upwards of 10 hours per assignment to grade a *single* code standard section (e.g. Spacing) before automating the process. This time can now be spent engaging directly with students by holding additional office hours or extended labs.

6 CONCLUSIONS AND FUTURE WORK

The authors propose two ideas concerning assessing student code projects in Computer Science. First, the authors 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. This recommendation is based on previously referenced statements by Ala-Mutka [1] and Yang [30], who present information suggesting a tangible association between code style and overall code quality. Computer Science departments seeking to replicate the success of Eastwood-Tidy are urged to adopt a code standard, such as the standard put forth here (see Standard I). However, the adopted standard need not be the same so long as it is comprehensive and well-specified. Well-specified rules (which some existing formatting tools like Black call "opinionated") ease implementation of Clang-Tidy rules and manual grading.

Second, when adopting such a code style guide, automated tooling such as Eastwood-Tidy should 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 [8] it is feasible to delegate grading of code standard adherence to Undergraduates, but doing so is not an effective use of time, especially when automated grading systems are available and possible.

The authors put forth Eastwood-Tidy, 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 stay with students as they continue their careers. It also reduces the workload on students and course staff in the software development and grading processes. It is worth noting that instructional assistants are still tasked with reviewing the output of automated tools to ensure no false positive or negative grades are assigned. For certain sections of the standard (e.g., whether variable names are appropriate), they are still more heavily used as well.

There is much 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. In addition, improved static analysis techniques would allow a coding standard to check for functionality errors that simple output-based automated test systems such as Vocareum employed heavily by university programs may support. In summation: code style is an exciting opportunity to reinforce student skills that are at present primarily left by the wayside.

The full source code, code standard text, testing framework, and documentation for Eastwood-Tidy is available at https://github.com/novafacing/eastwood-tidy. The code standard, edited significantly for brevity, is also included here. We would like to thank the many Teaching Assistants, as well as students, who helped in the development and refinement of this software. Additional thanks to Dr. Richard L Kennell (Purdue University) for early work in this area and contributions to the code standard.

7 CODE STANDARD

I. **Naming Convention**
- I.A Variable names should be in all lowercase
- I.B Use descriptive and meaningful variable names
- I.C Constants must be #defined, uppercase, ≥ 2 characters
- I.D All global variables begin with prefix g_

II. **Line and Function Length**
- II.A Lines over 80 characters are split up and indented
- II.B Each function must be under 240 lines

III. **Spacing**
- III.A One space must be placed between all structure and control flow keywords (ex while) and open parentheses, and before all open braces
- III.B One space precedes and follows all operators
- III.C One space follows internal semicolons and commas
- III.D #define directives must be grouped, left aligned, and surrounded by blank lines
- III.E No line should end with trailing whitespace
- III.F No spaces between function name and parameter list

IV. **Indentation**
- IV.A Place open braces on the same line as control flow keywords. Indent compound statements 2 spaces.
- IV.B Parameters should be on one line or aligned to the first parameter if line length is exceeded
- IV.C The while keyword of do-while loops should be on a line with a closing brace

V. **Comments**
- V.A Comments should be meaningful
- V.B Place comments above code except alongside declarations, else, and switch statements
- V.C Comments should be preceded and followed by a blank line and indented with surrounding code
- V.D Function name should be commented after function compound statement
- V.E Function header comments should be preceded and followed by a blank line

VI. **Multiple Logical Expressions**
- VI.A Logical sub-expressions should be surrounded with parentheses, except for the top-level expression

VII. **Headers**
- VII.A A comment should be placed above each function describing the interface and purpose of the function

VIII. **Header Files**
- VIII.A Every .c file should have a corresponding .h file
- VIII.B Header filenames should end in .h
- VIII.C All header files should have include guards, with defined symbol names ending in _H
- VIII.D All project header files should use relative paths and be descendants of the project directory
- VIII.E Includes should be ordered alphabetically with the corresponding header first, then local and global files
- VIII.F All included files should be included explicitly
- VIII.G Library includes must use angle braces

IX. **Defensive Coding**
- IX.A Return values must be guarded for error conditions
- IX.B FILE pointers must fclose-ed and set to NULL
- IX.C Pointers deallocated with free must be set to NULL
- IX.D Function parameters must be checked for errors
- IX.E Appropriate zero values should be used for each type

X. **Output Handling**
- X.A Error messages must be directed to stderr

XI. **Disallowed Statements**
- XI.A Do not use tabs for indentation
- XI.B DOS newlines should not be used
- XI.C Only one assignment should be made per expression
- XI.D Constant values should be defined using directives
- XI.E goto should not be used

XII. **Variable Declarations**
- XII.A Only one variable should be defined in one statement.
- XII.B All variables should be initialized where defined

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] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. [n.d.]. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2021-03-05) *(SIGCSE '21)*. Association for Computing Machinery, 1062–1068. https://doi.org/10.1145/3408877.3432430

[3] 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

[4] Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. [n.d.]. Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022-02-22) *(SIGCSE 2022)*. Association for Computing Machinery, 307–313. https://doi.org/10.1145/3478431.3499294

[5] Will Crichton, Georgia Gabriela Sampaio, and Pat Hanrahan. [n.d.]. Automating Program Structure Classification. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2021-03-05) *(SIGCSE '21)*. Association for Computing Machinery, 1177–1183. https://doi.org/10.1145/3408877.3432358

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

[7] Black Developers. 2022. *The uncompromising code formatter*. https://black.readthedocs.io/en/stable/

[8] 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

[9] GNU. 2022. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/

[10] GNU. 2022. *Guidelines for Diagnostics*. https://gcc.gnu.org/onlinedocs/gccint/Guidelines-for-Diagnostics.html

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

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

[13] Wouter Groeneveld, Dries Martin, Tibo Poncelet, and Kris Aerts. [n.d.]. Are Undergraduate Creative Coders Clean Coders? A Correlation Study. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022-02-22) *(SIGCSE 2022)*. Association for Computing Machinery, 314–320. https://doi.org/10.1145/3478431.3499345

[14] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[15] Abe Leite and Saúl A. Blanco. [n.d.]. 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* (New York, NY, USA, 2020-02-26) *(SIGCSE '20)*. Association for Computing Machinery, 44–50. https://doi.org/10.1145/3328778.3366921

[16] 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

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

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

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

[20] LLVM. 2022. *Clang C Language Family Frontend for LLVM*. https://clang.llvm.org/

[21] LLVM. 2022. *Clang-Format*. https://clang.llvm.org/docs/ClangFormat.html

[22] Logilab. 2001. *PyLint*. https://www.pylint.org

[23] 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

[24] 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

[25] TIOBE. 2022. *TIOBE Index*. https://www.tiobe.com/tiobe-index/

[26] Linus Torvalds. 2005. *Linux kernel coding style*. https://www.kernel.org/doc/html/v4.10/process/coding-style.html

[27] Vocareum. 2022. *Programming Lab*. https://www.vocareum.com/home/programming-lab/

[28] Eliane Wiese, Anna N. Rafferty, and Jordan Pyper. [n.d.]. Readable vs. Writable Code: A Survey of Intermediate Students' Structure Choices. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022-02-22) *(SIGCSE 2022)*. Association for Computing Machinery, 321–327. https://doi.org/10.1145/3478431.3499413

[29] 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

[30] 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