Survey of Tools and Settings for Introductory C Programming

Sunjae Park, Wentworth Institute of Technology

Sunjae Park is an assistant professor in the School of Computing and Data Science at Wentworth Institute of Technology, an engineering-focused institution in Boston. He received his undergraduate degree in Electrical Engineering from Seoul National University, and received a masters degree and PhD from Georgia Institute of Technology. His research interests are in program analysis and computer science education.

Survey of Tools and Settings for Introductory C Programming

Abstract

The C programming language is a language with a long history and used to write programs ranging from embedded device apps to operating systems. Although it's almost 40 years old, the language still regularly appears in the top programming languages in various programming language rankings. This means the language occupies an important part of many undergraduate engineering programs.

However, the language has many pitfalls that make it difficult for novices to learn. The language syntax is overly forgiving, accepting code that will be a compiler error in other languages. On the other hand, the language semantics is unforgiving, leading to security issues, crashes, or silent data corruption.

This paper surveys various tools and their settings that can ease the introduction. First, the paper introduces the development environment setup. Setting up a rich development and debugging environment for C can be complex, especially on Windows systems. Next, the paper surveys various compiler settings that are helpful for novice C programmers. There are around 200 compiler warning flags in the GCC compiler. Although most of these are useful, some of these warning flags are very aggressive and seems to confuse students. This paper reviews many of these and proposes a set that is most useful for students. Lastly, the paper surveys various C code analyzers and recommends those that are easiest to setup while still proving helpful.

Introduction

The C programming language has long been a staple in college computing education. Although Java and Python are popular languages, C is still a top programming language of instruction [1], [2]. Even if the introductory courses are taught in other languages, many programs still provide courses that teach the languages, typically in systems programming courses or operating system courses [3]–[5].

However, unlike Java or Python where there is a single authorative compiler, C programming is supported by many compilers, editors, and other tools. In addition, installing a C development environment has traditionally been challenging for Windows systems. As a result, some institutions opt for installing the C development environment in a server and have the students connect remotely. This means students are required to learn both the language and how to navigate in a remote (typically command-line) environment, which complicates learning.

This paper describes a development environment that is easier for students to get started in. The

basic premise of this paper is to identify freely available tools for C development on their personal computer (commonly a laptop computer). The tools should be cross platform to support students running a variety of operating systems including Windows, MacOS and Linux. They should also support code that is not tied to a specific development environment. They should also preferably provide provide quick feedback to the students, helping them develop correct code faster [6].

It starts by surveying what C development environments are available. It then discusses the compiler settings that the author found most helpful for students. Lastly, other tools are introduced that can help students in their development process.

Development Environment and Tools

In this section we discuss the development environments and tools that are available. Many of the tools introduced here are also popular among professional developers as can be seen from JetBrains's survey "The State of Developer Ecosystem 2023 [7]."

Compilers

The first things students need to install to develop in C would be a C compiler. Two compilers are well known for doing C development. The first is the GCC (GNU Compiler Collection), which is has a long history as the compiler of choice for many platforms [8]. The other compiler is clang, from the LLVM project [9]. Although the clang compiler has a shorter history than GCC, it has a reputation for providing compiler output and better diagnostics[10], [11]. In addition, as an entire compiler infrastructure, there are many tools built with clang as a basis, as we'll see in section . However, recent versions of both compilers have mostly caught up with each other, either option works well.

To install these compilers, MacOS and Linux users can use a package manager (such as Homebrew or apt/dnf) to easily install either of the two compilers.

Under Windows, which is used by the many students as their personal computer, this picture is more complicated. One path students can use is to use Windows Subsystem for Linux (WSL). WSL allows most development tools that's available under regular Linux, and so we'll consider WSL to be identical to Linux. However, this option is not available under older versions of Windows, and the installation is complicated.

Other option is to use native tools, where there are several options available. The first option is to use WinLibs, which provides various command-line tools to work with the GCC compiler. Once installed and student adds the ‘‘bin‘‘ folder to the PATH environment variable, then many standard C development tools (such as the C compiler or other code analysis tools) are made available. This option is the simplest option for Windows users. Another option is to install MSYS2. This program allows Windows users a Linux-like environment similar to WSL, with support for both compilers as well as shell, package manager, and other command-line tools.

Another popular compiler is included in Microsoft Visual Studio. This is a full-fledged Integrated Development Environment (IDE) developed by Microsoft, and very popular (4th in the developer ecosystem survey). The Community Edition is freely available for students. However, C programming language support has had limited support in this compiler [12]. It also has a

complicated installation process and complex user interface [13], and only supported on Windows (not on MacOS or Linux) [14].

Other than GCC and clang, there's also the TCC compiler and the Pelles C compiler. However, these compilers are not available on MacOS or Linux.

Development Environments

In addition to a compiler, students need a code editor is or some kind of development environment. If using a code editor, students will need to get used to running some kind of build runner (see Section) that will read the project description file (i.e. Makefile) and build the code. If using an integrated development environment (IDE), it should have support for a wide variety of project files. Popularity in a professional setting would be a positive (see the JetBrains survey [7]), but is not required.

One popular development environment for C programming is Visual Studio, which was discussed in Section . Although very sophisticated, this IDE can be difficult to use and is not cross-platform.

Another option is to use Code::Blocks. Code::Blocks is cross-platform IDE, and has been suggested as an IDE for novices [15]. The IDE does not provide a compiler/debugger (unlike Visual Studio) itself, but users can download the IDE with GCC included. Unfortunately, the IDE uses its own build system, and projects written in Code::Blocks are not portable to other development environments. It is not commonly used in a professional setting.

The Eclipse platform is a well known IDE initially developed for Java programming, but there is also a C development version. Like Java, it is cross-platform, and does not come with a compiler/debugger, but instead relies on GCC. It has similar disadvantages to Code::Blocks, in that it uses its own project format and does not popular among developers.

There is also Visual Studio Code. This is a code editor developed by Microsoft, and is not directly related to Visual Studio (other than the name). It is mostly open source and freely available and has cross-platform support. It is the most popular development environment for C programmers according to the developer ecosystem survey. There are many plugins that allow better language support than the vanilla editor. Due to this plugin nature, students can use the same editor for other languages as well.

GitHub Codespaces is Visual Studio Code, but wrapped inside a web browser. This allows students to use their browser to connect to a virtual machine running on GitHub's servers, within the Visual Studio Code setting. Arguably this will have the lowest barrier of entry, since students only need an account on GitHub and a web browser. Unfortunately this option is not free, and prevents assignments from using external libraries (like unit test libraries).

JetBrains provides a suite of development environments, and CLion, their C focused IDE is popular (2nd place). Although normally a paid product, students can take advantage of the educational license. CLion is a sophisticated IDE with many code analysis tools (although doesn't provide a compiler by itself), and supports standard project build setups. One additional benefit is that because CLion is a part of a suite of environments, students can easily move to a different member of the suite when developing for a different language (for example, when using Python in a

data science course).

Lastly, terminal editors such as vi, micro, or Emacs are also available. Unfortunately, using debuggers under these editors tend to be more difficult to use.

Table 1 summarizes the discussion in this section.

Table 1: Editors and IDEs

Name	Cross-platform	Compiler	Project-file Support
Visual Studio	Windows only	cl and clang	CMake
Code::Blocks	Yes	gcc	Own format
Eclipse CDT	Yes	gcc	Own format
Visual Studio Code	Yes	gcc and clang	CMake and partial make
CLion	Yes	gcc and clang	CMake and partial make
Terminal Editors	Yes	Any	Any

Project Setup

The make program, and it's configuration file $\mathrm{Makefile}$ has traditionally been the project setup tool of choice for C programs. It is well known and relatively simple to get started. However, using make for C development requires the knowledge of hidden rules and implicit targets (such as object file targets). Many IDEs also only provide partial support for Makefiles due to its complexity.

Recently, CMake is increasingly popular as a project manager for C programs. This tool uses files named $\mathrm{CMakeFiles.txt}$ in each directory to drive the build system, which can be either traditional make that was just discussed, or the ninja build system, which is a like a stripped down version of make.

Compared to using traditional make, CMake focuses more tightly C development, and has various options to generate configuration files for static analyzers, which we will discuss in Section . It is well supported by the development environments discussed in Section , such as Visual Studio, CLion, and Visual Studio Code. However, CMake configuration files ($\mathrm{CMakeLists.txt}$) perfer explicit rules which make it more verbose than make.

Unit Test Framework

Unit testing is used in many programming courses. Test-driven development is already a well established workflow for software development, so integrating unit tests into the curriculum is helpful for students. In addition, it can lead to improved student visibility in their progress, and a better sense of progress while working on assignments [16].

There are several unit test frameworks available that works with C programming. Some of the more popular ones are listed in Table 2.

The most popular framework according to the Developer Ecosystem Survey [7] is the Google Test framework. It has long been the most popular unit test framework, and can be easily installed on MacOS and Linux. Unfortunately, it's difficult to install on Windows systems. It's also written mainly for C++ programs, and although C and C++ are closely related languages, there are still

Table 2: Unit Testing Libraries

Name	Main Language	Description
Google Test	C++	Mature and well supported
catch2	C++	Well supported and full-featured
unity	C	C only and simple to understand

major differences. Writing moderately complicated unit tests require C++-specific features (such as classes), and can confuse students trying to understand what's going on in the tests.

Catch2 is another popular test framework. This is also mainly for C++ unit tests, and test authors can write even more clever code in this framework. Catch2 allows the programmer to arbitrary strings for names of unit tests (Google Test only allow names that follow the C function naming rules). It also allows for deeply nested tests to test all possible code paths. However, the added complexity can be confusing for novices, and the library increases compile time significantly (especially for the small programs that novices tend to write).

Lastly, the Unity test framework is a C-language only framework. It is simple and is written to be vendored directly into the project, so can be used in environments where installing library packages can be difficult (such as when using WinLibs or GitHub Codespaces). Because it's a simple library, test authors may find themselves repeating a lot of code or copy-pasting code, but this can make things easier for novices who mostly read unit test code, not write them.

Google Test and Catch2 are well supported by IDEs such as CLion or Visual Studio Code, enabling students to run test with a click of a button. Using Unity requires students use a terminal to run the test executables manually.

Configuration

In this section, we discuss the configuration settings for the tools discussed above. C being used in so many fields (from embedded development to high performance computing), there are many settings that can be set.

Compiler Warning Settings

As general rule, compiler warnings help prevent erroneous code, and it is important that the students get as much support as possible. On the other hand, some compiler warnings can be detrimental as well. It can be very confusing to figure out what compiler error messages mean [17]. Some warnings are only troublesome in certain corner cases, and some even have false positives, which can lead to alert fatigue and lead to students tuning them out [18].

Table 3 lists some of these flags that are available in both GCC and Clang. Note that the first two flags, `-Wall` and `-Wextra`, should always be enabled [19]. These two flags are enable common warnings that the compiler developers deem commonly useful. For each row, the name of the warning and a description is listed. Recommendation is also listed as well (‘‘Yes’’ for warnings that are recommended for use, ‘‘No‘‘ for those that are discouraged).

Variable shadowing occurs when a new variable has the same name as a another variable in the

Table 3: Recommended Settings for the Compiler

Name	Use?	Description
Wall	Yes	Basic warnings (should always enable)
Wextra	Yes	Additional warnings (should always enable)
Wshadow	Yes	If variables are shadowed
Wuninitialized	Yes	If the variable is used without initialization
Wfloat-equal	Yes	If floats are compared using the == operator
Wvla	Yes	If variable-length arrays are used
Walloca	Yes	If the alloca function is used
conversion	No	If numerical types are implicitly converted
sign-compare	No	If signed/unsigned values are compared
sign-conversion	No	If signed/unsigned values are converted
unused	No	If an identifier is unused
unused-parameter	No	If a function parameter is unused
unused-variable	No	If a variable is unused
format-nonliteral	Maybe	If a non-literal used for format string

outer scope. This will ”hide” the outer variable and can lead to confusing results, and using this warning can help students avoid this.

Uninitialized variables is one area where C differs from other languages. Many languages leave an uninitialized variable to contain some zero value. However, C does not, and uninitialized variables can contain garbage values. It can be very time consuming to debug buggy programs that use garbage values, and using this warning can help students avoid this.

Floating point numbers cannot be compared against correctly [20] in many cases. This is because decimal numbers cannot be accurately represented in binary floating point numbers (for example, 0.1).

Variable length arrays are a controversial topic in C programming. These are arrays that have a length component that's not a compile time constant, such as $\mathrm{int\ myarray[len]}$ where len is a variable. In C, this is can lead to buggy behavior [12] and its use is discouraged in many places. The vla flag generates a warning when this is used, and C Programmers should use create a large enough array or use heap memory. alloca has similar problems and is likewise discouraged.

The C programming language frequently allows numbers to be implicitly converted (i.e. int to float by dropping the decimal point). This can lead to incorrect behavior [21], [22]. However, the $\mathrm{conversion}$ flag frequently generates false positive warnings and can be a source of confusion for novices. ”signed int i” assigned to ”unsigned int j” triggers a warning. ”double d1” assigned to ”float f1” can also trigger a warning. More precise warnings, such as $\mathrm{float\text{-}conversion}$ still tends to generate false positives.

In many programming assignments, students are given a skeleton and are tasked to complete it. The assignment by it very nature will have variables and parameters that are unused, and the associated warnings should be disabled (i.e. by $\mathrm{-Wno\text{-}unused\text{-}parameter}$ and $\mathrm{-Wno\text{-}unused\text{-}variable}$).

Use of printf also has many challenges for novice programmers. Fortunately, both compilers enable many checks for incorrect printf use. One warning that may be disabled is the format-nonliteral warning. When the first argument to printf, the format string, is a non-literal string (such as a string read from a file), this can lead to security issues [23]. Some students get confused with this warning (especially those who used other languages before). However, the solution is simple enough that the warning may be kept.

Several other compiler flags related to warnings are also useful. When building C programs from the terminal, -Wfatal-errors prevents the compiler from proceeding, and prevents students from encountering dozens of errors from a single compile operation.

To make sure students pay attention to the warnings, -Werror may be used. However, this makes it even more important that only warnings that severely impact the correctness of the program should be enabled.

Sanitizer Settings

Some programming mistakes cannot be caught statically by the compiler. For example, an address index out-of-bounds error (for example, index -1) cannot be statically ruled out by a C compiler. For these types of issues, both GCC and Clang allow the use of sanitizers, which add runtime checks for errors in the code. The trade-off is slower code, but the sanitizers can be turned off when performance needs to be measured (for example, when comparing time complexity).

Address Sanitizer [24] can be enabled by passing GCC or clang -fsanitize=address. It is particularly useful for catching memory bugs that may arise. For example, a student by try to use a negative index in an array, try to dereference a pointer after its pointee has already been freed, or even forget to free. Address Sanitizer will print memory corruption issues immediately, or memory leaks after the program terminates.

Undefined Behavior Sanitizer can be enabled by passing GCC or clang -fsanitize=undefined. This tool can check for certain operations that can lead to problematic behavior, and *print the line where it happened*. For example, if the code encountered a null pointer, or tried to divide by zero, the program will crash. Unfortunately, unless the program was already running inside a debugger, the precise location of where the crash occurred isn't printed. This sanitizer on the other hand, will print the file and line, which is helpful for debugging.

Other Tools

In addition to the tools just discussed, there are other tools that are very useful in helping students identify problems in their code. One would be autoformatters such as *clang-format*. These tools automatically format the code, therefore reducing student confusion on what is "well formatted code." Both Visual Studio Code and CLion can run clang-format, or students can run it from the command-line interface.

Another tool would be static analysis tools. Static code analysis is similar to compiler warnings but involve more in-depth processing [25]. They are also more subjective, in that something noticed by clang-tidy is not necessarily a bug; instead, it can be considered "problematic." Some static analysis tools are cppcheck, GCC Static Analyzer, and clang-tidy. Since clang-tidy is better supported in the IDE's discussed in Section , we'll discuss this in more detail.

clang-tidy

clang-tidy is a standalone tool for running static analysis on the source code. Static code analysis is similar to compiler warnings but involve more in-depth processing [25]. clang-tidy also contains checks that are more subjective, in that something noticed by clang-tidy is not necessarily a bug; instead, it can be considered "problematic." For example, readability-else-after-return alerts the programmer if the if block has a return statement. In this case, the else may be removed. However, keep this else is not *in*correct, either.

Using clang-tidy checks are also a bit more complicated to use than than compiler warning. clang-tidy requires compilation database file (compile_commands.json). Fortunately, this can be generated by CMake (see Section) by the flag (-DCMAKE_EXPORT_COMPILE_COMMANDS). It can also be generated by command-line tools such as bear or compiledb.

Table 4 contains some checks are recommended and some that are discouraged. As discussed earlier, many of clang-tidy checks are more prone to false positives and more subjective, so instructors should start from a minimal subset and review the list of checks [26] to determine which ones to use. Many of them are also solely for C++ development, since C++ being a more complicated language results more checks.

Table 4: Selected Settings for clang-tidy

Name	Use?	Description
chained-comparison	Yes	Disallow comparisons like $x < y < z$
braces-around-statements	Yes	Require braces after if/else/for etc.
misleading-indentation	Yes	Require indentation to match code structure
misplaced-array-index	Yes	Require braces after if/else/for etc.
sizeof-expression	Yes	Prevent problematic arguments to sizeof
suspicious-string-compare	Yes	Prevent problematic strcmp checks
narrowing-conversions	No	If numerical types are converted to smaller ones
easily-swappable-parameters	No	If parameter types/names are similar

Discussion

The suggestions in this paper have been tried in several iterations by the author. Most of the suggestions were refined in a junior-level course on operating systems. Students in this course are to learn how to write simple applications in C, use system calls in their C programs, and write some algorithms used by the operating system. The course has weekly 2-hour in-class lab sessions, which allowed the instructor to get quick feedback on how the students were struggling with the development setup.

In the first iteration, students running Windows were instructed to install a hypervisor platform (such as VirtualBox or VMWare) to install a version of Ubuntu Linux on their personal machines. They were then instructed to install the gcc compiler, gdb debugger, clang-format and clang-tidy, and cmake and ninja build tools. For the editor, they were instructed to install Visual Studio Code.

For unit testing, the catch2 library was used since Visual Studio Code supports running catch2 tests in a graphical manner.

In this iteration, as many warning as possible were enabled, from Wall and Wextra to Wconversion and Wunused. Many clang-tidy checks were enabled as well, most notably bugprone- and readability- checks.

Most students were able to successfully complete the assignments. However, there were a few issues that were noticed. The biggest issue was that too many warnings were being triggered. For example, Wconversion flags any checks between signed and unsigned integers. Although *technically* this can be a problem for sufficiently large numbers, for most introductory C programming this is not a problem. For example, strlen returns an unsigned integer. This means the following code can lead to a compiler warning.

```
for(int i = 0; i < strlen("Hello"); i++)
```

Some other issues were the sluggishness of every operation. Running Ubuntu inside a virtual machine is taxing on the graphics card, and the slow build times of catch2 was a frequent complaint. Novice students tend to write relatively small programs, so the additional compile time can slow down their development cycle.

In subsequent iterations, students running Windows were instructed to use Windows Subsystem for Linux (WSL) instead of a hypervisor. Visual Studio Code (and CLion) both allow the native graphical application to access the WSL, so the previous semester's issues of UI frustration was diminished. The unit test library has also changed from catch2 to Google Test. Catch2 is a powerful library but takes a long time to compile.

Warnings were used more selectively as well. As noted earlier, students frequently ran into numerous warning messages that had limited impact in the code correctness (either because of false positives or because of limited scope). These warnings were removed in subsequent iterations. The current list of compiler warnings are: *-Wall -Wextra -Wshadow -Wuninitialized -Wvla -Walloca -Wno-sign-compare -Wno-sign-conversion -Wno-unused -Wno-unused-parameter*. For clang-tidy, we used those in Table 4.

Although some students still get confused with VLA warnings, the number of students struggling with the development environment has gone down, allowing them to focus effort on the idea instead.

Conclusion

This paper describes various tools and settings for C programming assignments. Novice students can struggle with learning the C programming language because of its relaxed syntax that is paired with unforgiving semantics. Code that is seemingly bug-free can crash in unexpected (to the student) ways.

It surveys the various tools that are actively used in the professional C development community. It discuses their pros and cons for a novice user, and discusses recommended settings. Lastly, the paper wraps up by discussing how it was used in several iterations and how students worked with the set up.

References

[1] R. M. Siegfried, K. G. Herbert-Berger, K. Leune, and J. P. Siegfried, ‘‘Trends of commonly used programming languages in cs1 and cs2 learning,’’ in *2021 16th International Conference on Computer Science and Education (ICCSE)*, 2021, pp. 407–412. DOI: 10.1109/ICCSE51940.2021.9569444.

[2] B. A. Becker and K. Quille, ‘‘50 years of cs1 at sigcse: A review of the evolution of introductory programming education research,’’ in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’19, Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 338–344, ISBN: 9781450358903. DOI: 10.1145/3287324.3287432. [Online]. Available: https://doi.org/10.1145/3287324.3287432.

[3] S. J. Matthews, T. Newhall, and K. C. Webb, ‘‘Dive into systems: A free, online textbook for introducing computer systems,’’ in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1110–1116, ISBN: 9781450380621. DOI: 10.1145/3408877.3432514. [Online]. Available: https://doi.org/10.1145/3408877.3432514.

[4] R. E. Bryant and D. R. O’Hallaron, *Computer systems: a programmer’s perspective*. Prentice Hall, 2011.

[5] R. C. Ferrao, I. Dos Santos Montagner, R. Caceffo, and R. Azevedo, ‘‘How much c can students learn in one week? experiences teaching c in advanced cs courses,’’ in *2022 IEEE Frontiers in Education Conference (FIE)*, 2022, pp. 1–8. DOI: 10.1109/FIE56618.2022.9962662.

[6] E. Allen, R. Cartwright, and B. Stoler, ‘‘Drjava: A lightweight pedagogic environment for java,’’ in *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, 2002, pp. 137–141.

[7] JetBrains s.r.o. ‘‘The state of developer ecosystem 2023.’’ (2023), [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2022/.

[8] W. von Hagen, *The Definitive Guide to GCC*. Apress, 2011, ISBN: 9781430202196. [Online]. Available: https://books.google.com/books?id=wQ6r3UTivJgC.

[9] C. Lattner, ‘‘Llvm and clang: Next generation compiler technology,’’ in *The BSD conference*, vol. 5, 2008, pp. 1–20.

[10] C. D. Bella. ‘‘Rfc: Improving clang’s diagnostics.’’ (2022), [Online]. Available: https://discourse.llvm.org/t/rfc-improving-clang-s-diagnostics/62584.

[11] easyaspi314. ‘‘Clang vs gcc vs msvc: Diagnostics.’’ (2018), [Online]. Available: https://easyaspi314.github.io/gcc-vs-clang.html.

[12] E. Dakeshov. ‘‘C11 and c17 standard support arriving in msvc.’’ (2020), [Online]. Available: https://devblogs.microsoft.com/cppblog/c11-and-c17-standard-support-arriving-in-msvc/.

[13] C. Reis and R. Cartwright, ‘‘A friendly face for eclipse,’’ in *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology EXchange*, ser. eclipse ’03, Anaheim, California: Association for Computing Machinery, 2003, pp. 25–29, ISBN: 9781450374705. DOI: 10.1145/965660.965666. [Online]. Available: https://doi.org/10.1145/965660.965666.

[14] A. Cangialosi. ‘‘Visual studio for mac retirement announcement.’’ (2023), [Online]. Available: https://devblogs.microsoft.com/visualstudio/visual-studio-for-mac-retirement-announcement/.

[15] A. J. Gonzalez, ‘‘Creating a simple c program, compiling, and executing it,’’ in *Computer Programming in C for Beginners*. Springer International Publishing, 2020, pp. 1–14, ISBN: 978-3-030-50750-3. DOI: 10.1007/978-3-030-50750-3_1.

[16] C. Desai, D. S. Janzen, and J. Clements, ‘‘Implications of integrating test-driven development into cs1/cs2 curricula,’’ *ACM SIGCSE Bulletin*, vol. 41, no. 1, pp. 148–152, 2009.

[17] B. A. Becker, P. Denny, R. Pettit, *et al.*, ‘‘Compiler error messages considered unhelpful: The landscape of text-based programming error message research,’’ in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR ’19, Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 177–210, ISBN: 9781450375672. DOI: 10.1145/3344429.3372508. [Online]. Available: https://doi.org/10.1145/3344429.3372508.

[18] C. Bravo-Lillo, L. Cranor, S. Komanduri, S. Schechter, and M. Sleeper, ‘‘Harder to ignore? revisiting {pop-up} fatigue and approaches to prevent it,’’ in *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, 2014, pp. 105–111.

[19] C. Wellons. ‘‘My favorite c compiler flags during development.’’ (2023), [Online]. Available: https://nullprogram.com/blog/2023/04/29/.

[20] R. Regan. ‘‘Why 0.1 does not exist in floating-point.’’ (2012), [Online]. Available: https://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/.

[21] D. Jovanovic. ‘‘C++: How to avoid implicit conversions.’’ (2021), [Online]. Available: https://dbj.org/cpp-how-to-avoid-implicit-conversions/.

[22] J. Turner. ‘‘Implicit conversions are evil and must go!’’ (2022), [Online]. Available: https://www.youtube.com/watch?v=T97QJ0KBaBU.

[23] H. Burch and R. C. Seacord. ‘‘Programming language format string vulnerabilities.’’ (2007), [Online]. Available: https://drdobbs.com/security/programming-language-format-string-vulne/197002914.

[24] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, ‘‘{Addresssanitizer}: A fast address sanity checker,’’ in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[25] S. Razmyslov. ‘‘What’s the difference between static analysis and compiler warnings?’’ (2014), [Online]. Available: https://pvs-studio.com/en/blog/posts/0274/.

[26] ‘‘Clang-tidy checks.’’ (2024), [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/list.html.