The Love/Hate Relationship with the C Preprocessor: An Interview Study

Flávio Medeiros[1], Christian Kästner[2], Márcio Ribeiro[3], Sarah Nadi[4], and Rohit Gheyi[1]

1 **Federal University of Campina Grande, Brazil**
2 **Carnegie Mellon University, USA**
3 **Federal University of Alagoas, Brazil**
4 **Technische Universität Darmstadt, Germany**

Abstract

The C preprocessor has received strong criticism in academia, among others regarding separation of concerns, error proneness, and code obfuscation, but is widely used in practice. Many (mostly academic) alternatives to the preprocessor exist, but have not been adopted in practice. Since developers continue to use the preprocessor despite all criticism and research, we ask how practitioners perceive the C preprocessor. We performed interviews with 40 developers, used grounded theory to analyze the data, and cross-validated the results with data from a survey among 202 developers, repository mining, and results from previous studies. In particular, we investigated four research questions related to why the preprocessor is still widely used in practice, common problems, alternatives, and the impact of undisciplined annotations. Our study shows that developers are aware of the criticism the C preprocessor receives, but use it nonetheless, mainly for portability and variability. Many developers indicate that they regularly face preprocessor-related problems and preprocessor-related bugs. The majority of our interviewees do not see any current C-native technologies that can entirely replace the C preprocessor. However, developers tend to mitigate problems with guidelines, even though those guidelines are not enforced consistently. We report the key insights gained from our study and discuss implications for practitioners and researchers on how to better use the C preprocessor to minimize its negative impact.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases C Preprocessor, CPP, Interviews, Surveys, and Grounded Theory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

The C preprocessor is a language-independent tool for lightweight meta-programming that fills a need, among others, for portability and variability. The preprocessor is widely used in practice. It is essentially used in all projects written in C and C++, including many well-known databases and operating systems. In academia, however, the preprocessor has received strong criticism since at least the early 90s. Researchers have criticized its lack of separation of concerns [5, 15, 25, 52, 57], its proneness to introduce subtle errors [6, 15, 33, 40, 44, 58], and its obfuscation of the source code [3, 6, 12, 42, 52]. Additionally, its complexity hinders tool support available in other languages, such as automating refactorings [20, 28, 29, 43, 66, 67]. Many studies have found bugs related to preprocessor use [1, 21, 29, 44, 60, 62]. The C preprocessor essentially has not changed since the 70s, but researchers have proposed several alternatives, such as syntactical preprocessors [8, 43, 68], aspect-oriented programming [2, 42] and various forms of metaprogramming. However, such alternatives have not been adopted in practice. Some projects adopt code guidelines such as "code cluttered with `#ifdefs` is difficult to read and maintain, don't do it" in the Linux kernel.[1] Although some tools could enforce such guidelines [6, 40, 56, 60], researchers show that they are not followed strictly in practice [12, 40].

Since developers continue to use the C preprocessor despite all criticism and research, this paper asks the basic question: **How do practitioners perceive the C preprocessor?** Do developers perceive similar problems as researchers indicate, or are problems exaggerated in the research literature? How do developers address potential problems and what kind of alternatives do they seek, if any? Are we possibly faced with a technology-transfer or education problem? Answering such questions provides guidance on research, tool building, technology transfer, and education.

To understand how developers perceive the C preprocessor, we interviewed 40 developers and cross-validated our results with (a) a survey among 202 developers, (b) results mined from software repositories, and (c) prior studies in this field. Complementing prior studies that analyzed how the preprocessor is used in *source code*, we actually talked to developers to solicit *perceptions and opinions*. We focus primarily on conditional compilation, because it is more controversial and error prone than lexical inclusion of files and macro expansion. In our research, we rigorously follow established empirical methods for interviews [17, 36], surveys [11], and text analysis [32]. Specifically, we follow a research method called grounded theory [4, 10]. We report the key insights gained in our interviews (and validated with the survey and other empirical data) and derive implications for practitioners and researchers.

Our results suggest that developers perceive the preprocessor as an elegant solution to handle portability and variability, but they are also well aware of the problems discussed in the academic literature. Developers typically report that they try to avoid preprocessor use or follow code guidelines to minimize problems, such as avoiding `#ifdef` directives inside function bodies and avoiding nesting of conditional directives beyond three levels. Most developers (over 80 %) particularly agree to avoid `#ifdef` blocks that do not align with the code structure (coined undisciplined annotations [40]) because they negatively impact code comprehension, maintainability and error proneness; we detected that actually only few developers introduced 85 % of all such cases.

Developers often deal with bugs related to preprocessor use. Most developers (67 %) agree these bugs are easier to introduce, and 74 % believe that they are harder to detect than other bugs. Our findings suggest that developers use inefficient testing strategies that normally do not detect bugs related to conditional compilation. Open-source developers instead rely on support from end-users to test the source code on different platforms and report bugs.

When asked for alternatives to preprocessor use, developers mainly discussed different encodings, C-language mechanisms, and build-system mechanisms. In contrast, alternatives such as syntactical preprocessors [8, 43, 68] and aspect-oriented programming [2, 42] have not been mentioned by any of our interviewees. Developers argued that new technologies would hinder adoption, since the C compiler is available for many platforms and the preprocessor is always there to deal with variability.

In summary, the main contributions of this paper are:

- We interviewed 40 developers to better understand how developers perceive the practical use of the C preprocessor and analyze the results using grounded theory.
- We cross-validated our interview findings by surveying 202 developers and comparing them with results from repository mining and prior studies.

[1] *Linux* kernel guidelines for patch submission in `Documentation/SubmittingPatches`.

- We discuss results and implications of our study for researchers and practitioners.

2 State of the Art

The preprocessor is widely used in practice, in essentially all projects written in C and C++. It is executed during the compilation process and performs three interacting tasks: (a) it lexically includes files (`#include`), (b) it expands macros (defined with `#define`), and (c) it conditionally excludes part of the source code depending on which and how macros are defined (`#ifdef`, `#if`, etc). All three functions of the C preprocessor have been criticized.

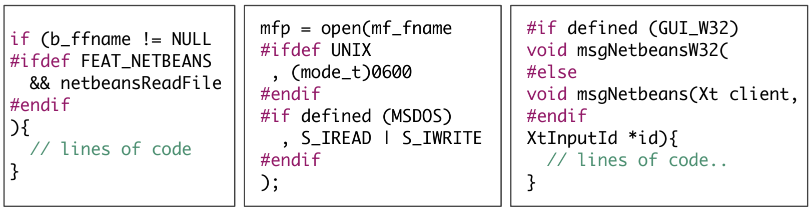
- Lexical inclusion causes large amounts of I/O operations during compilation and slows down the build process. For example, an average file in the *Linux* kernel includes over 300 header files [29]. There is movement in the C community towards a proper build system to replace `#include` directives [23].
- Lexical macros allow all kinds of potential problems [12] since they have no notion of structure, hygiene, or capture avoidance that advanced macro systems support [9,16, 31,43,68]. Developers avoid these problems by following certain patterns when defining macros [12], which are broadly adopted and also checked by a number of static analysis tools [56]. In addition, C++ introduced several language features to replace common uses of preprocessor macros [34,46].
- Since conditional compilation removes code before compilation, it causes compilers and many other analysis tools to see only parts of the code. It has been criticized as limiting separation of concerns, as obfuscating the code, as being error prone, and as preventing tool support, as we will discuss next. Interactions of conditional compilation with lexical inclusion and macro expansion make it even harder to reason about the preprocessor execution.

In the following, we discuss challenges induced by the C preprocessor (especially conditional compilation, because inclusion and macro expansions are relatively well understood) and available mitigation strategies, as they are discussed in the research literature. These challenges guided us in the design of our study to analyze whether and how the perception of developers differs from that in the research literature.

2.1 Readability and separation of concerns

Many research studies criticized the preprocessor regarding its limited separation of concerns and code obfuscation, which make maintenance and code comprehension difficult [6,12, 19,29,40]. In particular, when conditional directives are used at fine granularity and are strongly scattered, it can be difficult to follow the control flow logic [3,52]. Such source code is sometimes referred to as the "`#ifdef` hell" by developers [42]. Long and deeply nested conditional compilation directives also can make it difficult to see when specific code fragments are included [5,33,52]. Several researchers have proposed aspect-oriented programming as an alternative [2,42], where optional code would be separated into distinct code artifacts and woven together at compile time, but we are not aware of any adoption beyond some research projects.

A specific practice that has been discussed in detail is the use of *undisciplined annotations:* conditional compilation directives that do not align with the code structure as illustrated in Figure 1. Undisciplined annotations have been related to error proneness [12,29,40,44], hindered code understanding and maintainability [6,12], and limitations in tool support (see below) [6,18,19,51]. An empirical study by Liebig et al. [40] revealed that most conditional



```
if (b_ffname != NULL
#ifdef FEAT_NETBEANS
  && netbeansReadFile
#endif
){
  // lines of code
}
```

```
mfp = open(mf_fname
#ifdef UNIX
  , (mode_t)0600
#endif
#if defined (MSDOS)
   , S_IREAD | S_IWRITE
#endif
);
```

```
#if defined (GUI_W32)
void msgNetbeansW32(
#else
void msgNetbeans(Xt client,
#endif
XtInputId *id){
  // lines of code..
}
```

Figure 1 Real code snippets taken from *Vim* with undisciplined annotations.

compilation directives in 40 open source C projects are disciplined, but 15.6 % of all `#ifdef` blocks do not align with the code structure.

2.2 Combinatorial explosion and parsing unpreprocessed C code

Conditional compilation decides which code fragments to include (including other preprocessor directives) depending on the values of macros. The number of possible preprocessed variants explodes exponentially with the number of macros involved in `#ifdef` and similar directives. C projects often have a large number of conditional directives depending on many macros; for instance, which parts of the *Linux* kernel are compiled depends on more than 12 thousand macros [38, 60].

A separate analysis of every possible preprocessed variant simply does not scale in any but the smallest systems. A typical strategy to cope with the combinatorial explosion is through sampling, for example, by analyzing representative or large variants with most conditional code included. For more systematic sampling, researchers have proposed combinatorial testing strategies [26, 49] and strategies that maximize configuration coverage [60]. Sampling is inherently incomplete though and may not discover issues occurring only in few variants due to interactions or complex `#if` conditions.

Some researchers have started to investigate tools that can parse unpreprocessed code and preserve all compile-time choices during the analysis. While earlier tools used unsound heuristics or supported only specific usage patterns of the preprocessor (e.g., requiring disciplined annotations) [6, 19, 51], more recent tools as *TypeChef* [29, 41] and *SuperC* [22] can accurately parse and analyze unpreprocessed C code, covering all configurations. In the product-line community, such analyses are called *family-based analyses* [64].

2.3 Error proneness and guidelines

Previous studies discussed the error-prone characteristics of the preprocessor [12, 15, 58] and found many bugs related to conditional compilation [1, 12, 21, 30, 44, 53, 60, 61], ranging from dead code to syntax and type errors and to behavioral issues and memory leaks. Spencer and Collyer [58] argue that many macro combinations are tested and often do not even make sense. Others argue that the simplicity of the C preprocessor enables developers to make ad-hoc extensions instead of restructuring the code, which leads to poor code quality and bugs related to preprocessor usage [6, 15].

Code guidelines have been developed to prevent certain common problems, e.g., undisciplined annotations or scope issues with macros [12, 40, 56]. Even though some of them can be enforced automatically by analysis tools [40, 56], research shows that such code guidelines are often but not strictly followed [12, 40].

2.4 Difficulty of developing tool support and syntactic preprocessors

Finally, preprocessor directives also make the development of tool support more difficult [22, 29, 40]. Even simple tasks as removing obsolete macros or identifying dead code require sophisticated analyses [6, 61]. Developing refactoring engines for C code is extremely challenging due to the need to parse unpreprocessed code (possibly with undisciplined annotations) and the need to deal with macro expansion [18, 20, 39, 67]; it is challenging even when conditional compilation is not considered [50, 59].

Many academic proposals for preprocessor alternatives are driven by a desire to provide better tool support and analysis. For example, *ASTEC* is a syntactic preprocessor that enables precise refactoring [43]. Several other syntactic preprocessors or related environments have been proposed [8, 9, 13, 27, 65, 68]. Some researchers propose means to refactor existing C code to alternative implementations [2, 43, 65] or at least undisciplined to disciplined annotations [19, 45, 55]. We are not aware of any adoption of these alternatives in practice though.

All prior studies on the C preprocessor that we are aware of were based on conceptual arguments or evidence extracted from software repositories. Our study is designed to elicit the *perception* of developers by talking to them.

3 Research Method

The goal of our research study is to analyze the strengths, drawbacks, and alternatives to the C preprocessor, as perceived by C developers. We specifically collect information about the C preprocessor that cannot be observed by analyzing only artifacts as in previous studies. We performed this research study primarily by interviewing developers and cross-validating our results with survey questions, other information from software repositories, and related studies. In this section, we give an overview of our research method. Details can be found in the appendix.

For this research, we combine several empirical research methods, including interviews, surveys, and mining software repositories. Empirical research methods allow us to investigate how human developers think and behave. We study not only the outcome of the development process, but assess also their opinions and perceptions. If not conducted carefully, empirical research can result in biased and superficial results. However, whole communities of researchers have investigated how to perform empirical studies that reduce biases and enable reliable and reproducible research despite potentially vague research materials. For example, following strict protocols and documenting steps and research results when analyzing transcribed interviews can mitigate many biases that researchers might otherwise introduce. In addition, cross-validating results from different sources is essential. This way, results complement and confirm each other and form a more reliable bigger picture. In this research, we strictly followed established research methods and cross-validated our results across several sources and with prior research results, as we will explain.

3.1 Research Questions

Our research is motivated by the mismatch between the critique that the C preprocessor has received from academics [6, 12, 19, 29, 40, 58] and the number of alternatives [2, 8, 42, 43, 68] on the one side and the broad use in practice on the other side. Specifically, we raise the following research questions:

RQ1. Why is the C preprocessor still widely used in practice?

RQ2. What do developers consider as alternatives to preprocessor directives?
RQ3. What are common problems of using preprocessor directives in practice?
RQ4. Do developers care about the discipline of preprocessor annotations?
We present the results for each research question separately in Sections 4–7.

3.2 Research Strategy

We performed our research in three phases, designing three studies. In the first phase, we analyzed the literature and identified the research questions stated above (see also Section 2). In the second phase, we performed semi-structured interviews with 40 developers (Study 1). In the third phase, we cross-validated our interview findings by conducting a survey among developers contributing to open source C projects (Study 2; 202 responses), mining data from 24 software repositories (Study 3), and comparing our results with prior research results.

3.3 Corpus

For all three studies, we use a corpus of 24 open source C systems. With the revision history of the systems in the corpus, we identified candidate interviewees and survey participants, and we studied technical aspects. We selected the systems in the corpus based on prior corpus studies on the C preprocessor [12, 38], covering a range of different domains and sizes (2.6 thousand to 7.8 million lines of code). We selected only projects for which we could find developer contact information in commits. The corpus includes the following projects: *Apache, Bash, Bison, Cherokee, Dia, Flex, Fvwm, Gawk, Gnuchess, Gnuplot, Gzip, Irssi, Libpng, Libsoup, Libssh, Libxml2, Lighttpd, Linux, Lua, M4, Mpsolve, Rcs, Sqlite,* and *Vim*.

3.4 Study 1: Interviews

We started our study by interviewing developers on how they perceive the C preprocessor. To reduce any potential bias and to make our study replicable, we followed the established exploratory research method *grounded theory* [4, 10]. We performed *semi-structured* interviews [17, 36], which are informal conversations where the interviewer lets the interviewees express their perception regarding specific topics. To elicit not only the foreseen information, but also unexpected data, we avoided a high degree of structure and formality and, instead, used open-ended questions. To cover the topic broadly, our questions evolved during the interview process based on gained insights [4, 10]. We followed standard guidelines regarding how to perform interviews [17, 36]. For example, we explained the purpose of the interviews, we provided clear transitions between major topics, we did not allow interviewees to get off topic, we allowed interviewees to ask questions before starting the interview, and we scheduled the interviews beforehand.

The interviews were grounded in research questions RQ1–4. We typically started an interview by asking developers about their experience with the C preprocessor and then tried to cover 4-6 different topics. The topics evolved during the interviews, and we asked different topics to different developers based on their background and answers. This is a standard approach to cover a topic broadly and qualitatively. Questions included *'In which situations do developers use conditional directives?'*, *'How do developers test different macro combinations in their code?'*, and *'What do developers think about directives that split up parts of C constructions?'*; see the appendix for a complete list. In addition to these questions, we used code snippets to ask developers concrete questions about code to encourage them to give more concrete answers. For each interviewee, we searched through the code repositories and selected code snippets related to that specific developer. We sent such snippets by email before the scheduled interview.

We performed 10 phone and 30 email interviews. We initially contacted developers via email presenting some information about our project and asked them to participate. We encouraged developers to perform phone interviews, but we also provided the alternative to answer our questions via email. When necessary, emails interviews involved back and forth conversations (i.e., a dialogue between researcher and participant). We sent at least one additional email with further questions in 19 (63%) out of the 30 email interviews we conducted. This and the fact that we cover the same questions in both phone and email interviews allows us to discuss them together as interviews, and not separately as phone and email interviews. To analyze the interview transcripts, we again followed established research methods: coding the answers, analyzing keywords, organizing them into concepts and categories, and writing memos [10]. We met weekly to discuss the memos and noticed that interviewees progressively started to give similar answers, a situation called *saturation* [10]. At this point, we considered the topic sufficiently clear and focused on other topics that needed further elaboration. The specific coding outcome is listed in the appendix.

We selected participants for the interviews from active developers in the 24 projects of our corpus. By mining the repositories, we identified the top 10 % active developers in each projects that regularly use conditional compilation (ranked by code churn). We sent emails to 213 open-source developers, and 32 (15 %) participated in our interviews. Even though many open-source contributors expressed that they primarily worked in industrial projects, we additionally explored whether interviewees recruited from industrial projects would provide additional insights. After reaching out to our contacts (convenience sampling), eight developers from Brazilian companies accepted to participate in our interviews. Most of our 40 interviewees self-identified as having at least 5 years of experience and many worked both within open-source and industrial contexts. Our developer selection is biased toward developers with experience with conditional complication, which we counteracted however by cross-validating our results with a survey of a broader population. See the appendix for a characterization of the interviewees. In our result presentation, we refer to individual anonymized participants as P1–P40.

3.5 Study 2: Survey

Whereas our interviews are designed to elicit qualitative insights into practices and reasons, our survey is designed to collect quantitative data from a large population. We designed the survey after completing and evaluating the interviews. It is a standard research approach to first perform qualitative investigations to identify relevant questions and subsequently perform a survey to explore them quantitatively in a larger population [24, 47].

With the survey, we explored topics that were unclear from the interviews or where we wanted additional quantitative data. We performed an online survey to reach more developers and again followed common guidelines for that research method [11]. For several questions, the survey included code snippets to make questions more concrete. We mention the survey questions while discussing our results in Sections 4–7. Details on the survey, including the exact questions, can be found in the appendix.

To select participants for our survey, we aimed at reaching a broader audience of developers with different levels of experience regarding conditional directives usage. We randomly sampled from all developers that contributed to the 24 projects in our corpus, excluding our interviewees. We sent emails to 3,091 developers and 202 (6.5%) filled out our survey.

3.6 Study 3: Mining undisciplined annotations

To investigate the issue of undisciplined annotations further, one of the most controversial and criticized issues in the literature, we mined software repositories to analyze different versions of the source code and statically detected undisciplined annotations. Specifically, we analyzed each commit in 14 projects of our corpus. We considered only projects with at least two active developers to compare their programming style regarding undisciplined annotations. We used a modified version of Liebig's *Cppstats* tool [40]. With this tool, we identified all commits that introduced undisciplined annotations, data which we analyzed grouped by developer. Subsequently, we interviewed four developers regarding their reasons for introducing specific undisciplined annotations. See the appendix for details.

3.7 Threats to validity

We selected interviewees by sending email to developers and only those interested in the topic participated in our study. From 40 interviews, even though they cross 24 projects of different sizes and domains and 3 companies, it is difficult to generalize results. Nonetheless, we alleviated these threats by cross-validating with a survey of a larger population. Our survey could be filled out in around 10-15 minutes, which encouraged more developers to participate. Code snippets used in our survey might be misunderstood by developers or might conflate multiple issues; that is, related results can only be interpreted in the context of these snippets. To detect undisciplined annotations, we used *Cppstats* [40], which is based on heuristics and may miss-classify a small number of annotations, but we expect that this does not affect the bigger picture collected across multiple projects.

In the following, we report the main findings of our study, structured by our four research questions. At the end of each research question, we summarize our main findings and the corresponding data sources used.

4 RQ 1: Why is The C Preprocessor Still Widely Used in Practice?

The C preprocessor has been heavily criticized in previous research, which raises the question of why it is still used in practice (RQ1). To fully answer this question, we need two pieces of information. The first is whether developers are actually aware of these (academic) criticisms, and the second is the set of scenarios in which developers find the C preprocessor useful. If developers are aware of the potential problems, but still use the C preprocessor, this suggests that there are cases in which using the C preprocessor is still the preferred or even the only available alternative. However, to identify such cases, we first need to understand the various situations in which the C preprocessor is used.

4.1 Developers' Awareness of C Preprocessor Criticism

We find that developers are aware of the criticism the C preprocessor has received, but they still believe that it is an elegant solution to handle variability and overcome portability problems, if properly used *(P1-P3, P18, P22, P23, P26)*. As one developer (*P39*) explains: *"Every feature of any technology can be abused or misused. When used appropriately, the use of preprocessor directives is not a problem."* That said, many developers *(P1-P3, P5-P8, P19, P20, P22-P26, P30-P33)* are aware that they must follow code guidelines to minimize problems related to code comprehension, maintainability, and error proneness (C preprocessor problems are discussed in more detail in Section 6).

4.2 Usage of the C Preprocessor

Our discussion with developers reveals the following fives cases in which they use the C preprocessor.

- *Portability.* Despite being from different domains, many of the systems we studied need to support multiple platforms and operating systems. The C preprocessor is perceived as a convenient way to ensure the system's portability across these different environments. For example, developers often use conditional directives to check settings of operating systems, platforms, compilers, and library versions *(P1-P3, P6, P17-P25, P27)*. Based on these settings, developers use certain macros, types, and header files that may only be available when using a specific operating system or compiler. For example, it is not possible to include *Windows* specific headers such as `windows.h` when compiling the source code on *Linux* or *Mac OS*. In addition to handling platform-specific header files, portability also involves checking for specific system constraints as well as making use of platform-specific functionality during implementation *(P1, P3, P18, P19, P21, P24, P27)*. For example, in some operating systems, such as *GNU Hurd*, there is no imposed limit on overall file name length, as there is on *Windows*.
- *Variability.* Developers often repeat that they use conditional directives to provide *optional features* or to *select between alternative implementations*. For example, one participant (*P4*) describes his use of variability as follows: *"I use conditional directives to remove parts of the library I do not need, since it makes the binary code much smaller."* Reducing binary size may influence developers' decision in using macros to represent optional functionality (i.e., *features*). The `DEBUG` feature is one extreme example of this, which was mentioned frequently by developers. `DEBUG` is a common feature developers use to print messages along the source code to understand what is going on during execution *(P21, P22, P23)*. Since `DEBUG` may not be useful for end-users, developers guard debugging code with the corresponding macro such that end-users can exclude it from the binary code during compilation. Several developers also state that they commonly use conditional directives to support alternate implementations (*P13, P27, P38-P40*). For example, in *Libssh*, developers can choose between different cryptographic libraries such as *Libcrypt* or *Libcrypto*, depending on the characteristics of the cryptographic algorithms they want to use. They find that the C preprocessor provides a convenient way to switch between such libraries at compile-time
- *Code Optimization.* Some developers explain that, apart from excluding unnecessary functionality, they also explicitly use conditional directives to optimize the code for performance or size *(P3, P4, P40)*. Interviewees explain that they often do not trust that all compilers will properly optimize their code. Thus, in some cases, developers take the task of optimizing the code into their own hands by implementing known code optimizations after checking for compiler name and version at compile-time using the C preprocessor. For example, the *Gcc* compiler offers some *GNU Extensions* such as type discovery and zero-length arrays. Developers explain that they want to make use of such optimizations if they are aware of their availability as this allows them to actively make the binary code smaller and faster.
- *Code Evolution.* A few developers state that they also often use conditional directives during the introduction of new code versions related to critical functionality *(P27, P28, P39)*. In this context, they introduce new implementations inside conditional directives, but they remove the previous version only when the new version is stable. They explain that by using conditional directives, they can switch between the old and new implementations for testing purposes.

```
#ifdef DEBUG
#define DEBUG_MSG printf
#else
#define DEBUG_MSG (format, args...) ((void)0)
#endif
// Developers do not need to check #ifdef DEBUG multiple times..
DEBUG_MSG ("message..");
```

Figure 2 Using function-like macros to avoid avoid code duplication (checking if `DEBUG` is defined multiple times) and to support encapsulation.

- *Language Limitations.* Several developers mention using conditional directives because of the limitations of the C language *(P6, P14-P16, P20, P36-P38).* For example, they use `#ifdef` checks to avoid multiple inclusion of header files. Such header guards (or include guards) are probably one of the few applications of the C preprocessor that is accepted by critics [58].

Some developers also mentioned using macros and function-like macros to avoid code duplication and to encapsulate frequently-changing code *(P20, P39, P40).* This way, developers need to only change the definition of the macro instead of changing all occurrences in the code. For example, Figure 2 shows how function-like macros can be used to define the behavior of `DEBUG_MSG`. While avoiding duplication and supporting encapsulation are not specific to the C language, using the C preprocessor is perceived as a convenient way to change function definitions at compile-time instead of at run-time. Previous studies [35, 46] considered the replacement of preprocessor macros with new features and idioms in the C++ programming language.

4.3 Discussion

We observed that the answers in our interview data reached a saturation point that is why we did not include this research question in our survey. This is also supported by the fact that many of the cases of C preprocessor usage we find (apart from the rare case of supporting code evolution) align with those found in previous work. For example, Ernst et al. [12] observed that portability accounts for 37 % of the use of conditional directives in the systems they examined, while include guards account for 6.2 %. They also found frequent usage of inline functions or function-like macros. Ernst el al. also argue that in order to eliminate some of the preprocessor usage, developers must be confident that the compiler will perform the necessary code optimizations. Our interviews support this and further suggest that, even after more than a decade, developers still lack this confidence in compiler optimizations.

Summary 1

Developers are aware of the criticism the C preprocessor receives, but still use it in the following situations: (1) supporting portability, (2) supporting variability, (3) providing code optimizations, (4) supporting code evolution, and (5) overcoming language limitations.

Data Sources: Interviews (Study 1) and Prior studies [12, 40, 54, 58]

5 RQ 2: What do Developers Consider as Alternatives to Preprocessor Directives?

We have seen that developers are aware of the problems and risks of using the C preprocessor but still have several use cases for which they need the C preprocessor's functionality. While researchers have proposed alternatives [8, 9, 43, 68], we wanted to see which alternatives developers are aware of or would recommend. We asked whether developers have thought about alternatives to the C preprocessor. We did not ask about specific alternatives or tools, because it was apparent that they usually would not be familiar with them. When we asked for preferences, we were only comparing different ways of using the C preprocessor. Our main goal with this question was to identify perceived alternatives, not to judge existing ones.

We generally received three kinds of answers: suggestions to use the C preprocessor in specific ways (guidelines on how to structure the code), suggestions to use in-language runtime mechanisms instead of compile-time mechanisms of the preprocessor, and arguments that the preprocessor cannot be replaced. Equally important is that none of our interviewees mentioned alternative preprocessors, aspect-oriented programming, or other metaprogramming solutions suggested by researchers. In the following, we discuss the three kinds of answers we received, cross-validated with survey findings.

5.1 Guidelines for Structuring Code

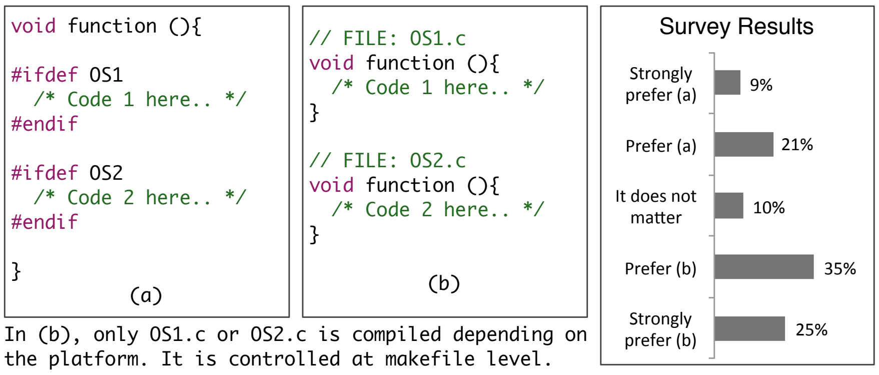
The first common suggestion to avoid using the C preprocessor is to separate alternative and optional code on the function, file and directory structure level *(P3, P8, P9, P14, P18, P24)*. For functions, the idea is to define alternative implementations of a function in separate files and to use the build system and the linker to choose the desired one. Figure 3b shows an example of this. Similarly, grouping related files in the same directory can also move compilation control to the build system, i.e., the whole directory will be compiled or not. Such structuring of the code means that no preprocessing is necessary within files. Additionally, the code structure is portable and requires no special tools. Nonetheless, one developer *(P15)* cautions that structuring the code in this way may leave it more difficult to comprehend. It is also difficult to deal with similar functions and code duplication if developers do not use helper functions for the common code.

The answers we received align with previous academic discussions [48, 58], but did not reach a saturation point in our analysis. Therefore, we asked a larger population of developers whether they prefer this code structuring strategy. In the survey, we present the two equivalent code snippets shown in Figure 3, asking developers which one they prefer based on a five-point Likert scale. We find that 30 % prefer to use conditional directives inside function bodies (i.e., Figure 3a), while 60% prefer to use different functions to solve portability concerns (i.e., Figure 3b). The remaining 10% of respondents had no preference between both options.

5.2 Alternative In-language Runtime Mechanisms

Another alternative that was suggested frequently during our interviews is the use of run-time variability binding (i.e., C `if` Statements) instead of compile-time binding with the preprocessor (i.e., `#ifdef` directives). An example of this is shown in Figure 4b.[2] Many

[2] While the decision here is still made statically, it could also be loaded from command-line options.



```
void function (){

#ifdef OS1
  /* Code 1 here.. */
#endif

#ifdef OS2
  /* Code 2 here.. */
#endif

}
            (a)
```

```
// FILE: OS1.c
void function (){
  /* Code 1 here.. */
}

// FILE: OS2.c
void function (){
  /* Code 2 here.. */
}

            (b)
```

In (b), only OS1.c or OS2.c is compiled depending on the platform. It is controlled at makefile level.

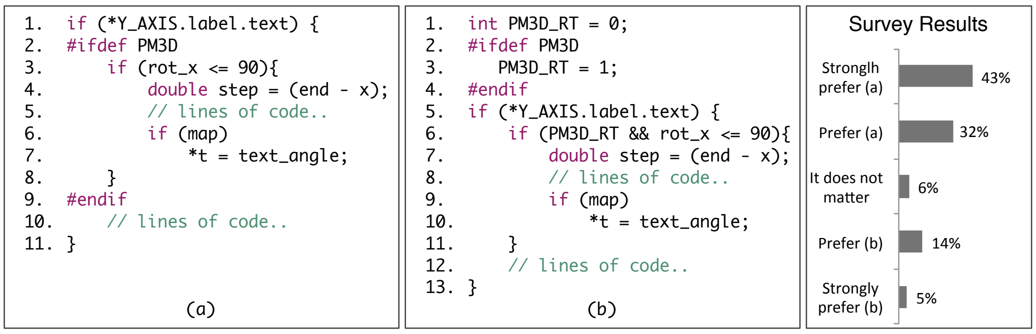
Figure 3 **(a)** Using preprocessor conditional directives inside function bodies to solve portability concerns. **(b)** Using different functions to avoid conditional directives inside function bodies.

developers state that they prefer to solve variability at run-time, when possible, since it is more flexible *(P1-P4, P6, P23, P40)*. One developer (*P23*) illustrates on this, saying: *"If something can reasonably be done without the preprocessor, I choose [to do it] that way. [Once the binary is there,] it is much more flexible to enable functions at run-time or with a configuration file than having to recompile the project again."*

To achieve such run-time variability, interviewees suggest using variables and enumerators instead of macros with constant values. They also suggest using inline functions to optimize the source code instead of function-like macros. However, developers also caution that run-time variability, e.g., the use of global variables and enumerators, is not thread-safe in C, and that using run-time variability is not possible in some cases. For instance, when runtime checks are not feasible due to performance reasons. One developer clarifies that run-time checks would cause performance overheads when checking for debugging mode, for example. This developer explains that when your goal is to process millions of I/O operations in the Linux kernel, for example, having run-time (i.e., C `if`) debug checks to verify certain assumptions would prevent you from scaling. However, developers still need a mechanism to easily verify assumptions when checking for code correctness, and they suggest that this can be cheaply achieved using the C preprocessor at compile-time.

Since developers' comments about run-time checks were not entirely consistent, we use the survey to see the preference of a broader population. This time, we present the two code snippets shown in Figure 4. In Figure 4a, we use conditional directives, while in Figure 4b, we use run-time variability with C `ifs`. We again ask survey participants to indicate which style they prefer using. Surprisingly, 75 % mentioned that they prefer to use conditional compilation directives (i.e., Figure 4a) while only 19 % prefer to use run-time variability (Figure 4b). The remaining 6 % of developers did not have a preference.

Based on the results of our interviews, we expected a higher percentage of developers to prefer using run-time variability in the survey. Accordingly, we went back to our interview data to see if we can find supporting reasons for why this might be the case. We find that developers might be inclined to use `#ifdefs` instead of `if` checks because of the following reasons. First, as stated by developer *P1*, when using conditional directives, it is easier to see the optional code. In other words, it is clear that the block of code from lines 3 to 8 is optional in Figure 4a. Second, developers *P3* and *P4* mentioned that by using variables instead of macros, developers do not know whether the compiler will optimize the source code. For instance, if the developer does not define `PM3D` in Figure 4b, variable `PM3D_RT`



```
1.  if (*Y_AXIS.label.text) {
2.  #ifdef PM3D
3.      if (rot_x <= 90){
4.          double step = (end - x);
5.          // lines of code..
6.          if (map)
7.              *t = text_angle;
8.      }
9.  #endif
10.     // lines of code..
11. }
```

(a)

```
1.  int PM3D_RT = 0;
2.  #ifdef PM3D
3.     PM3D_RT = 1;
4.  #endif
5.  if (*Y_AXIS.label.text) {
6.      if (PM3D_RT && rot_x <= 90){
7.          double step = (end - x);
8.          // lines of code..
9.          if (map)
10.             *t = text_angle;
11.     }
12.     // lines of code..
13. }
```

(b)

Figure 4 **(a)** Code snippet of *Gnuplot* with preprocessor conditional directives within function bodies. **(b)** Using local variables to avoid checking preprocessor macros throughout the code.

is always zero, i.e., `false`, and the block of code from lines 7 to 10 becomes unreachable. Developers argue that compilers may perform optimizations to remove such dead code, but there are no guarantees (i.e., this is compiler specific and depends on the compiler settings). Additionally, a few developers mention that some compilers may issue warnings about such unreachable code or about cases where the `if` condition would always be true which they find annoying *(P3, P29)*.

5.3 No Perceived Alternatives

Several developers mention that they have not thought about alternatives to the C preprocessor *(P17, P19-P21, P25-P28)* and that they are comfortable with using the C preprocessor for the purposes previously discussed. As stated by one developer (*P40*): *"Preprocessor directives can be used to remove the most tedious and error-prone parts of programming. It [is] also the only C-native way to conditionally compile when run-time checks are unacceptable [due] to performance [overheads]. There are no alternatives to the C preprocessor for this type of usage without using some tool outside the language."* Similarly, additional developers mentioned that in some cases, they really need to remove parts of the source code *(P1, P6, P23, P27)*. Otherwise, the code will not compile because of platform-specific parts that have not been removed. This leads them to argue that it does not matter which alternative one comes up with, one will need the C preprocessor at some point for such a platform-specific conditional compilation problem. Additionally, developers expressed concern that new technologies that replace the C preprocessor are likely not going to be present in all compilers *(P1, P6, P20)*. This shows hesitation to adopt third-party tools or alternate technologies (e.g., aspect-oriented programming [2, 42] or new macro languages such as ASTEC [43]) because of portability concerns.

Summary 2

Developers do not see any current technologies that can entirely replace the C preprocessor. However, some developers routinely use alternate coding styles such as dividing functionality into separate files or functions (preferred by 60 %) and using run-time checks instead of ***`#ifdef`*** *checks (preferred by 19 %).*

Data Sources: Interviews (Study 1), Survey (Study 2), and Prior studies [48, 58]

6 RQ 3: What are Common Problems of Using Preprocessor Directives in Practice?

We now try to understand what problems, if any, do developers face while using the C preprocessor. We find that developers' comments generally align with the problems raised in the research literature. Specifically, developers mention the following problems: (1) dealing with preprocessor-related bugs, (2) testing an exponential number of configurations, and (3) difficulty with understanding code with too many `#ifdefs`.

6.1 Preprocessor-related Bugs

A number of developers confirm that bugs related to conditional compilation occur frequently *(P7, P18, P23, P35-P37)* or at least sometimes *(P8-P10, P13, P14, P27)*. Our interviewees list different types of bugs related to the use of preprocessor directives, such as: incompatible macro selection *(P17)*; macros resulting in erroneous control flow *(P20, P22, P26)*; incorrect macro expansion *(P9)*; misspelled macro names *(P22, P23)*; missing variables and functions such as defining a variable in optional code and using it in mandatory code *(P13)*; type errors *(P8, P18, P23)*; syntax errors like missing control flow tokens, e.g., opening and closing brackets *(P24)*; linking problems *(P24)*; behavioral changes due to macro interactions *(P1, P9)*; memory and resource leaks, memory corruption, and race conditions *(P14)*; and incorrect use of `#else` and `#elif`. For instance, `#else` clause incorrectly treating some configurations, or use of `#elif` without an `#else` at the end to treat the default case *(P14)*.

Some interviewees *(P3, P20, P27)* argue that it is hard to deal with a high number of different macro combinations, which may ease the introduction of bugs. Developer *P1* points out that code that does not compile is easy to deal with, but the runtime bugs are the harder ones to detect. On the other hand, some developers *(P8, P10, P13, P20)* mention that even to detect simple compiler errors, someone has to compile the source code using the specific configuration that contains errors which is not that easy. The types of bugs developers mentioned align with various results from previous studies [7, 14, 37, 44, 61, 62] that we discussed in more detail in Section 2. Additionally, the fact that dealing with macro combinations is one of the sources of bugs is consistent with the findings of Iago et al. [1].

Since the data from our interviews is qualitative, we used our survey to quantify the frequency of C preprocessor-related bugs. We also asked developers about the difficulty of introducing preprocessor-related bugs when compared to other types of bugs as well as difficulty of detection. We present the results in Figure 5a-d, which can be summarized as follows. Even though developers believe that preprocessor-related bugs do not occur very frequently (Figure 5a), they find that they are slightly more critical than non-prerprocessor related bugs (Figure 5b) and that they are easier to introduce (Figure 5c) and harder to detect (Figure 5d). Our survey findings are consistent with our prior work [29] that parsed all code of a *Linux* kernel (release `x86`, only) and did not find any syntax errors. On the other hand, when the same authors analyzed *BusyBox*, they only found a few type and linker errors that they reported to developers and which were fixed in subsequent releases [30]. This supports our findings that such types of errors may be rare, but are still important to fix nonetheless.

Both our interview and survey results suggest that, similar to researchers [1, 29, 40], practitioners perceive the C preprocessor as error-prone. However, developers did not mention having any tools that help them with avoiding such errors. Our findings suggest that we need further research on developing tool support to minimize bugs related to preprocessor usage and finding ways to make such tools attractive for developers to use.

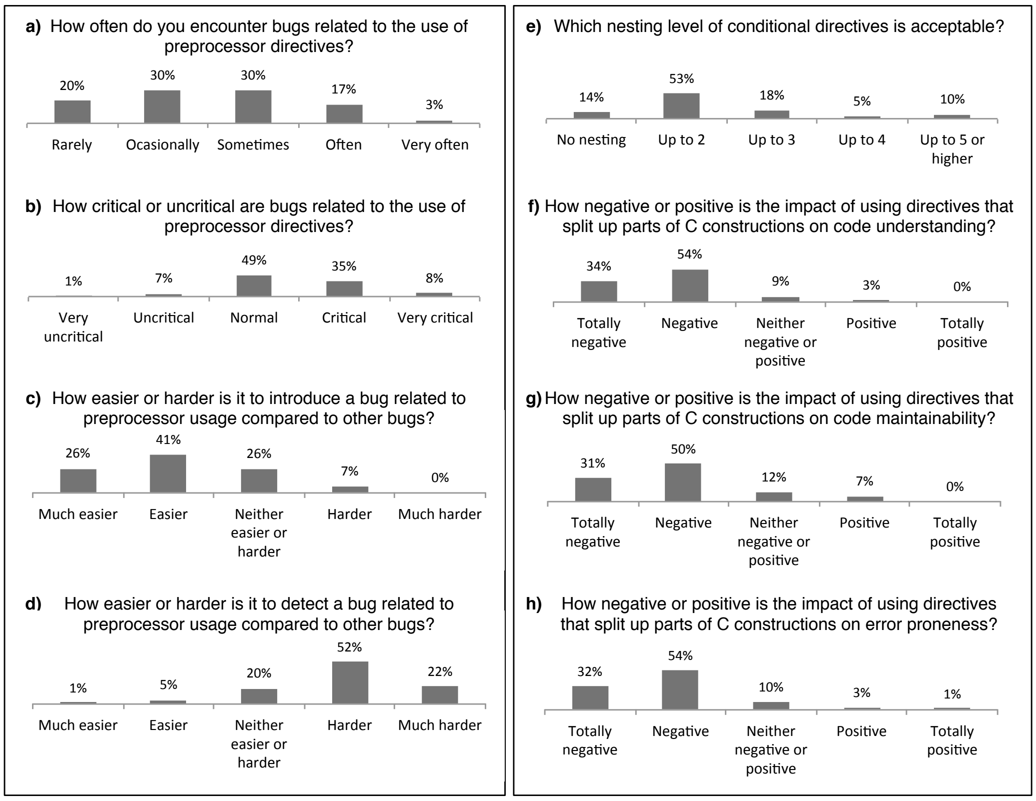


Figure 5 Results of our survey to quantify some findings of our interviews.

6.2 Combinatorial Testing

Developers explain that another problem with using the C preprocessor is dealing with combinatorial explosions. As mentioned by developer *P19*, the use of conditional directives makes the code hard to test and debug since it increases the testing matrix. The number of configurations to test grows exponentially when developers add new preprocessor macros in `#ifdefs`. Assuming that there are n optional and independent macros, developers have 2^n different configurations. Furthermore, developers explain that they also need to consider different compilers, operating systems, and platforms, which is time-consuming and makes automation difficult. For these reasons, many developers *(P1-P6, P9, P17, P19, P20, P22, P23, P25)* mentioned that they normally do not test all different macro combinations due to time and resource constraints. They explain that they do not have an easy way to test all possible combinations.

Many developers *(P9, P17, P19, P20, P22, P23, P25)* mention that what happens in practice is that they check only a few configurations of the source code. Furthermore, some developers *(P11, P18, P24)* check only the default configuration on their own machine with all optional macros active. However, a few developers *(P1, P37)* mention that they additionally consider different platforms besides their own. They say that by compiling the source code with two or three different compilers and using 32 and 64-bit platforms, they are comfortable enough that the code is portable. Similarly, some developers *(P19, P20, P22)* said that they select specific configurations to test by setting different macro combinations manually.

This variation in testing style tells us that there is no systematic way to fully test such systems. We find that developers *(P2, P26)* often rely on end-users to test the source code using different platform configurations. Developer *P26* explains this as follows: *"I check whatever combinations I can, and some combinations can only be tested on systems to which I have no access. I rely on others to help out or just cross my fingers."* Developer *P2* echoes this, also stating that he heavily relies on his user base to report back errors. This result aligns with a recent study on testing in the *Eclipse* platform by Greiler et al. [24]. Some developers *(P4, P10, P13)* realize that they use a narrow testing strategy and perceive it as a problem, expecting to find additional bugs if they are able to test more configurations. For example, one developer (*P26*) tell us: *"I do not find bugs related to preprocessor usage by running tests, but when running the tests with different combinations of macros."*

We find that testing in industry and open-source projects are different. While our open-source interviewees repeatedly mention testing only a few configurations and relying on user testing, industry developers *(P31, P32, P38)* mention that they test the source code on all supported platforms with all macros active. Additionally, some industrial developers *(P33-P36)* state that they check all combinations and platforms supported. This difference can be explained by the lack of community involvement in the industry context and that the number of used configurations tends to smaller (companies can restrict the number of supported configurations).

To overcome some of the challenges above, several developers *(P8-P12, P14, P31-P34)* mention that they use style checkers and static-analysis tools that often help them avoid bugs. This is especially true in industry projects. Our interviewees used the following tools: *Checkpath*, *Vera++*, *Coverity*, *Cppcheck*, *Valgrind*, *Coccinelle*, and *Lint*. Other developers *(P7, P13)* mentioned that they use at least *Gcc* with all warnings active. However, these tools consider only one configuration at a time, after preprocessing. Thus, these tools do not focus on bugs related to preprocessor usage. Some tools, e.g., *Cppcheck*, try to deal with many configurations by activating one macro at a time and performing the analysis several times, which is time-consuming. *Coccinelle* also handles variability to some extent by building a control-flow graph per function, where statement-level `#ifdefs` are taken into consideration. During the interviews, only one *Linux* developer (*P9*) mentioned a research tool, *Undertaker* [63], that can detect dead `#ifdef`-guarded blocks. However, developers did not mention any of the research tools that could analyze all configurations, such as *TypeChef* [29] or *SuperC* [22].

6.3 Code Comprehension

Our interviews suggest that many developers find it hard to understand code that is filled with `#ifdefs`. Developers *(P1, P5)* mentioned that the mixing of C code and directives interrupts the code logic since they are two independent languages. Developers *(P1, P5, P6, P19, P21, P25, P26)* believe that this mixing can obfuscate the source code making it harder to read, comprehend, and maintain since it is difficult to determine which parts of the code are going to be compiled under which conditions. For example, some developers *(P1, P5, P23, P24)* complain about the use of fine-grained directives within function bodies. It requires the analysis of control flow structures (`if`, `while`, `switch`, and `goto` statements) as well as `#ifdef`, `#ifndef`, `#if`, `#else`, and `#elif` directives. In addition, it becomes harder to understand the control flow, more difficult to check whether opening and closing brackets match, increases code complexity, and may lead to bugs. Additional developers *(P10, P18, P20, P24)* confirm this, saying that they often avoid preprocessor directives because of readability problems. One developer (*P6*) specifically comments on this, saying *"My main*

```
if (user_callbacks == NULL) {
#ifdef HAVE_PTHREAD
  callbacks=&ssh_pthread;
}
#else
  return SSH_ERROR;
}
#endif
```
(a)

```
if (user_callbacks == NULL) {
#ifdef HAVE_PTHREAD
  callbacks=&ssh_pthread;
#else
  return SSH_ERROR;
#endif
}
```
(b)

Figure 6 **(a)** Undisciplined annotation. **(b)** One possible equivalent/disciplined version.

problem is that [if] there are macros 7 layers deep[,] I don't understand them."

We used our survey to gain further insight into the impact of C preprocessor directives on code comprehension and maintainability as shown in Figure 5e. We find that 14 % of our interviewees state that they prefer to avoid nesting preprocessor conditional directives altogether, 53 % do not mind using nesting up to level 2, and only 18 % find that three levels of nesting is still acceptable. Note that only a few developers find that deep nesting levels (i.e., those beyond three) are acceptable. Overall, implicitly, 85 % of the developers see that nesting levels beyond three should be avoided. This aligns with previous work that finds that the average nesting level across 40 analyzed C systems is approximately 1 [38].

Summary 3

Developers face three preprocessor-related problems: (1) preprocessor-related bugs (do not appear often, but are perceived as more critical than other bugs), (2) combinatorial testing (conditional directives increase number of configurations to test), and (3) code comprehension (due to deep nesting of `#ifdefs`).

Data Sources: Interviews (Study 1), Survey (Study 2), and Prior studies [1,7,14,29,37,38, 40,44,61,62]

7 RQ 4: Do Developers Care About the Discipline of Preprocessor Annotations?

The feasibility of introducing undisciplined annotations (see Section 2) is one of the most criticized aspects of the C preprocessor [6,12,18,19,22,29,40,44], which is why we dedicate a separate research question for it. Our goal is to find out whether developers also view undisciplined annotations as problematic.

The majority of interviewees *(P1, P5, P17-P28, P32)* agree that the use of preprocessor directives to encompass individual tokens or parts of C syntactical units impacts the quality of code negatively. Such developers emphasize that they would not use undisciplined annotations because they hinder source code readability *(P5, P17, P18, P22, P25, P26, P28)*, obfuscate control flow *(P1, P24)*, and make the code difficult to evolve and maintain *(P20, P22, P23)*. One developer (*P20*) elaborates on this, saying: *"I avoid this kind of directives, they make the source code hard to understand and maintain. My gut feeling keeps screaming possible bugs when I'm faced with a code like that."* Along the same lines, one of these developers recommends to discourage or disallow undisciplined annotations through code guidelines to avoid the aforementioned problems *(P26)*. Another developer *(P30)* stated that code guidelines are important for the homogeneity of the project and that he often asks contributors to rewrite patches to follow the guidelines.

Despite the criticisms we received from most interviewees as shown above, some developers

(P4, P22, P31) mention that they would still use undisciplined annotations in very specific cases. In such cases, they would also document the code to let others understand their reasoning. Furthermore, some developers (*P5, P7*) are reluctant to change undisciplined annotations once they exist. For instance, one developer (*P5*) states that: *"One thing is to not fix what is not broken. The problem is that to refactor a code, you have to understand [it]. If you do not understand [it], it is not easy to refactor. Many developers would say: I am not going to touch that."* Developer *P39* mentioned that while he believes it is good to fix undisciplined annotations, it has a very low priority. It is worth noting that none of the developers mentioned using tools to enforce disciplined annotations or identified a lack of tool support in general.

To generalize our findings, we use the survey to quantify the impact of undisciplined annotations on code understanding, maintainability, and error proneness as shown in Figure 5f-h. Our results show that developers generally agree that undisciplined annotations have a negative impact on code understanding (88 %), maintainability (81 %), and error proneness (86 %). However, in a previous study, Schulze et al. [55] could not establish significant differences between disciplined and undisciplined annotations from a program comprehension perspective in a controlled experiment. Nevertheless, they observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor annotations. Additionally, although developers see undisciplined annotations negatively, other researchers [40] detected that almost 16 % of conditional directives are undisciplined annotations.

To investigate this gap between developer preferences and perceptions and the reality in the code base, we performed an additional analysis of software repositories to identify the reasons why developers introduce undisciplined annotations. By analyzing 14 software repositories of our corpus, we detected that only 21 (7 %) out of 299 developers introduced almost 85 % of all undisciplined annotations we found in the software repositories. When we tried to contact these developers, some got defensive and excused their use of undisciplined annotations. For instance, one developer argues that, *"The code was actively rewritten at the time, and it often happens that first drafts of an idea ends up in poor code."*

Figure 6a presents an example of an undisciplined annotation introduced in one of the C projects we examined. When we discussed this code fragment with its author, the author mentioned to prefer the equivalent code snippet in Figure 6b as a replacement. Another developer who we contacted about undisciplined annotations stated to use such annotations to avoid cloning the source code as well as compiler warnings. Figure 7a presents the undisciplined annotation introduced by this developer. When discussing the code, the developer showed us the alternative in Figure 7b that clones the source code (see lines 5 and 8) and another that generates compiler warnings as shown in Figure 7c, both of which seemed unacceptable to that developer. In this latter case, variable `failed` is always `true` when macro `USE_NTLM_AUTH` is not defined. In addition, this developer mentioned that since the undisciplined annotation in the original code was not repeated in many places, this minimizes potential problems. Such examples show that there may be situations where developers would prefer to use undisciplined annotations. In a previous study, We proposed alternatives to discipline annotations without cloning the source code [45]. However, they did not take compiler warnings into consideration. According to our data, compiler warnings seem to be a problem that may hinder the adoption of syntactical preprocessors despite of the existence of compiler attributes to ignore specific warnings.

```
#ifdef USE_NTLM_AUTH
if (priv->sso_available) {
  conn->state = SSO_FAILED;
} else {
#endif
  conn->state = NTLM_FAILED;
#ifdef USE_NTLM_AUTH
}
#endif
```
(a)

```
#ifdef USE_NTLM_AUTH
if (priv->sso_available) {
  conn->state = SSO_FAILED;
} else {
  conn->state = NTLM_FAILED;
}
#else
conn->state = NTLM_FAILED;
#endif
```
(b)

```
boolean failed = TRUE;
#ifdef USE_NTLM_AUTH
if (priv->sso_available) {
  conn->state = SSO_FAILED;
  failed = FALSE;
}
#endif
if (failed){
  conn->state = NTLM_FAILED;
}
```
(c)

Figure 7 (a) Undisciplined annotation. **(b)** Alternative that clones code. **(c)** Alternative that generates compiler warnings.

Summary 4

Most developers agree that undisciplined annotations impact code understanding, maintainability, and error proneness. However, there are cases where developers use undisciplined annotations to avoid code clones and compiler warnings.

Data Sources: Interviews (Study 1), Survey (Study 2), Mining Repositories (Study 3), and Previous work [40, 45, 55]

8 Concluding Remarks

We performed interviews with 40 developers about their perceptions of the C preprocessor, used grounded theory to analyze the data, and cross-validated the results with data from a survey with 202 developers, repository mining, and previous studies. Our study makes a step toward understanding how developers perceive the C preprocessor, enabling new perspectives on practices, guidelines, tools and technology transfer, and possible research directions. We found that the C preprocessor is still widely used in practice mainly to solve portability and variability problems. Developers are aware of problems and follow guidelines and adopt runtime variation, but they see no alternatives to the preprocessor; developers are skeptical about using new technologies outside the language. In addition, preprocessor-related bugs are perceived as less frequent, but easier to introduce, harder to fix, and more critical than other bugs. We also find that more than 80 % of developers do not like to use undisciplined annotations because they negatively impact code maintainability, comprehension and error proneness.

Our study has several implications for practitioners and researchers:

1. **Guidelines and enforcement.** Practitioners use guidelines to avoid common well-known pitfalls of the C preprocessor. In addition to information mined from repositories [12, 40, 44], our study provides a first step to develop guidelines grounded in data and taking into account developer preferences and acceptance. For example, we found strong evidence that developers support separating portable code at functions or files granularity, avoiding deep nesting of `#ifdef` directives, and avoiding undisciplined annotations. Whereas many developers routinely use analysis tools and respond to their warnings, there are currently few tools to enforce preprocessor-related guidelines systematically.
2. **Quality assurance.** Configurations are rarely tested systematically or even exhaustively, and developers perceive this as a problem. Restriction on configurations (especially in industrial projects) and community involvement (especially in open source projects) are current pragmatic strategies to cope with large configuration spaces (similar to plug-in systems [24]). We argue that systematic sampling [49, 60] and family-based analyses [64] are promising directions.

3. **Tool design and technology transfer.** The cross-platform availability with every compiler is an asset of the C preprocessor that developers are not willing to give up and that limits the adoption of alternative tools. Changes at the standardization level seem more effective, as done with module systems [23] and C++ extensions to replace the need for lexical includes and macros [34, 46]. Researchers might want to make their tools more attractive to developers by taking their perspective and needs into account. Low awareness of research tools indicates a communication problem. External tools that automatically detect guideline violations (such as undisciplined annotations) and propose fixes (e.g., refactorings) can likely have a larger impact on practice and simplify work for downstream analysis or refactoring tools that can take advantage of limited usage patterns.

The purpose of this study is not to design new language mechanisms, but we discussed preferable alternatives, many of which have been adopted in modern languages. Still, the large amount of existing C code (and also Fortran and C++ code which frequently use the C preprocessor) and the difficulty of analzying and porting it due to the particularities of the C preprocessor make it worth studying the problem both from a technical perspective and also as a technology transfer problem. While the results do not directly generalize beyond the C preprocessor, it demonstrates perceptions of developers that might be of interest also to other language and tool designers.

Acknowledgements We thank all participants of our interviews and surveys for their time and open discussions. We are grateful for constructive feedback from Sven Apel, Jörg Liebig, Janet Siegmund, and Iago Abal on earlier drafts of this paper. This work was supported by CNPq grants 573964/2008-4 (INES), 306610/2013-2, 477943/2013-6 and 460883/2014-3, NSF grant CCF-1318808, NSERC CGS-D2-425005 and the DFG Project E1 within CRC 1119 CROSSING.

References

1 Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 Variability bugs in the Linux Kernel: A qualitative analysis. In *Proceedings of the International Conference on Automated Software Engineering*, ASE. IEEE/ACM, 2014.

2 Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proceeding of the International Conference on Aspect-Oriented Software Development*, AOSD. ACM, 2009.

3 Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. *Electronic Communications of the European Association for the Study of Science and Technology*, 2008.

4 Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4), 2011.

5 Michalis Anastasopoules and Critina Gacek. Implementing product line variabilities. In *Proceedings of the Symposium on Software Reusability*, SSR. ACM, 2001.

6 Ira Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Procedings of the Working Conference on Reverse Engineering*, WCRE. IEEE, 2001.

7 Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the International Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2008.

8 Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE. ACM, 2010.

9 Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM. ACM, 2002.

10 JulietM Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1), 1990.

11 Don A. Dillman, Jolene D. Smyth, and Leah Melani Christian. *Internet, Phone, Mail, and Mixed-Mode Surveys: The Tailored Design Method.* Wiley, 2014.

12 Michael Ernst, Greg Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12), 2002.

13 Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transaction on Software Engineering and Methodology*, 21(1), 2011.

14 David Evans. Static detection of dynamic memory errors. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI. ACM, 1996.

15 Jean-Marie Favre. Understanding-in-the-large. In *Proceedings of the International Workshop on Program Comprehension*, IWPC, 1997.

16 Matthew Flatt. Composable and compilable macros:: You want it when? In *Proceedings of the International Conference on Functional Programming*, ICFP. ACM, 2002.

17 Uwe Flick. *An Introduction to Qualitative Research.* SAGE Publications, 2014.

18 Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE, 2002.

19 Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a C program. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2005.

20 Alejandra Garrido and Ralph E. Johnson. Embracing the c preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12), 2013.

21 Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE. IEEE, 2011.

22 Paul Gazzillo and Robert Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI. ACM, 2012.

23 Doug Gregor. A module system for the C family, 2012. Remarks by Doug Gregor at The sixth general meeting of LLVM Developers and Users.

24 M. Greiler, A. van Deursen, and Margrete-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2012.

25 Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. C/C++ conditional compilation analysis using symbolic execution. In *Proceeding of the International Conference on Software Maintenance*, ICSM. IEEE, 2000.

26 Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the International Software Product Line Conference*, SPLC, 2012.

27 Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2008.

28 Christian Kästner, Sven Apel, and Martin Kuhlemann. A model of refactoring physically and virtually separated features. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2009.

29 Christian Kästner, Paolo Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2011.

30 Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA. ACM, 2012.

31 Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the International Conference on LISP and Functional Programming*, LFP. ACM, 1986.

32 Klaus H. Krippendorff. *Content Analysis: An Introduction to Its Methodology.* SAGE Publications, 2014.

33 Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the International Conference on Software Engineering*, ICSE. IEEE, 1994.

34 Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2012.

35 Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 2012.

36 Steinar Kvale. *InterViews: An Introduction to Qualitative Research Interviewing.* SAGE Publications, 1996.

37 David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium.* USENIX Association, 2001.

38 Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of International Conference on Software Engineering*, ICSE. ACM, 2010.

39 Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2015.

40 Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD. ACM, 2011.

41 Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, FSE. ACM, 2013.

42 Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the European Conference on Computer Systems*, EuroSys. ACM, 2006.

43 Bill McCloskey and Eric Brewer. Astec: A new approach to refactoring C. In *Proceedings of the European Software Engineering Conference and International Symposium on Foundations of Software Engineering*, ESEC/FSE. ACM, 2005.

44 Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating Preprocessor-Based Syntax Errors. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, GPCE. ACM, 2013.

45 Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. A catalogue of refactorings to remove incomplete annotations. *Journal of Universal Computer Science*, 2014.

46 Christopher A. Mennie and Charles L. A. Clarke. Giving meaning to macros. In *Proceedings of the International Conference on Program Comprehension*, ICPC. IEEE, 2004.

47 Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2014.

48 Sarah Nadi and Ric Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8), 2014.

49 Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computer Surveys*, 43(2), 2011.

50 Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring C with macros. In *Proceeding of the International Symposium on the Foundations of Software Engineering*, FSE. ACM, 2014.

51 Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*. Springer, 2009.

52 T. Troy Pearse and Paul W. Oman. Experiences developing and maintaining software in a multi-platform env. In *Proceedings of the International Conference on Software Maintenance*, ICSM. IEEE, 1997.

53 Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2014.

54 Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2011.

55 Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, GPCE, 2013.

56 Robert C. Seacord. *The: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley, 2014.

57 Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS. ACM, 2007.

58 Henry Spencer and Geoff Collyer. `#ifdef` considered harmful, or portability experience with C news. In *USENIX Annual Technical Conference*, 1992.

59 Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11), 2003.

60 Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000 `#ifdefs` issue. In *USENIX Annual Technical Conference*, 2014.

61 Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *Proceedings of the Conference on Computer systems*, EuroSys. ACM, 2011.

62 Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale sys-

tem software. *International Journal on Software Tools for Technology Transfer*, 14(5), 2012.

63 Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: finding zombie features in the linux kernel. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, FOSD, 2009.

64 Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1), 2014.

65 Federico Tomassetti and Daniel Ratiu. Extracting variability from C and lifting it to mbeddr. In *Proceedings of the International Workshop on Reverse Variability Engineering*, REVE, 2013.

66 Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE. ACM, 2006.

67 Marian Vittek. Refactoring browser with preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR. IEEE, 2003.

68 Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of Internatinoal Conference on Programming Language Design and Implementation*, PLDI. ACM, 1993.