The Love/Hate Relationship with the C Preprocessor: An Interview Study (Artifact)*

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

This appendix presents detailed information about the research methods we used in the study, subject characterization, grounded theory process that we followed strictly, and the survey we performed in the study. It provides helpful data for understanding the subtler points of the companion paper and for reproducibility.

1998 ACM Subject Classification D.3.4 Processors
Keywords and phrases C Preprocessor, CPP, Interviews, Surveys, and Grounded Theory
Digital Object Identifier 10.4230/DARTS.1.1.7
Related Article Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi, "The Love/Hate Relationship with the C Preprocessor: An Interview Study", in Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015), LIPIcs, Vol. 37, pp. 495–518, 2015.
http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.495
Related Conference 29th European Conference on Object-Oriented Programming (ECOOP 2015), July 5–10, 2015, Prague, Czech Republic

1 Experimental Design

The goal of our study is to increase our understanding about how developers perceive the C preprocessor in practice. This study aims at collecting information about the C preprocessor that cannot be observed by analyzing only artifacts as in previous studies. We performed this study primarily by interviewing developers and asking survey questions. This appendix provides detailed description of the experimental design we followed.

Research Questions

Our study focuses on four research questions. We observe that a number of research studies and practitioners have criticized the use of the C preprocessor due to its negative impact on code quality [2, 5, 7, 11, 8]. However, a misconception might exist since the preprocessor is still widely used in practice to handle variability and portability [13]. This motivates us to investigate the following main research questions:

* 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.

- **RQ1.** Why is the C preprocessor still widely used in practice?
- **RQ2.** What do developers consider as alternatives to preprocessor directives?
- **RQ3.** What are the common problems of using preprocessor directives in practice?
- **RQ4.** Do developers care about the discipline of preprocessor annotations?

To answer our research question we combine insights from three studies (interviews, survey, mining software repositories) with data from related work. Our studies were performed on a single corpus of subject systems. We searched for developers' information, such as name and email address, in the 24 software repositories. We recruited our interviewees (Study 1) and survey participants (Study 2) mostly from developers of these projects. We performed further analyses on undisciplined directives (Study 3) on a subset of the corpus. Overall, we interviewed 40 developers and cross-validated our interview findings by using data from a survey with 202 developers, software repository mining, static analysis, and results from previous studies.

Corpus of 24 projects

In Table 1, we list the projects of our corpus with corresponding information about the projects' size and domain. We select our corpus inspired by previous work [5, 10] that analyzed the C preprocessor usage and quantified the number of *undisciplined* annotations. We selected projects from previous studies to be able to cross-validate our results and considered only projects that use C as the primary programming language. We selected projects from different domains, such as operating systems, databases and web servers. The size of our selected projects also varied ranging from 2.6 thousand to 7.8 million lines of code. We selected only projects for which we could find developers' information (e.g., name and email address) in commits. For our third study, we used a subset of our corpus, consisting of 14 projects, as indicated in the last column of the Table 1. We selected only projects with at least 2 active developers. An active developer has high code churn along the commit history. In the projects we considered for Study 3, there was a significant gap between the code churns of active developers and other non-frequent contributors.

Study 1: Interviews

We started our qualitative study by interviewing developers regarding our main research questions. To reduce any potential bias and to make our study replicable, we followed established research methods. Specifically, we adopted an exploratory research method, *grounded theory* [3, 1], to understand how developers perceive the practical use of the C preprocessor. We performed *semi-structured* interviews [9, 6], which are informal conversations where the interviewer lets the interviewees express their perception regarding specific topics. During the interviews, we were interested in qualitative instead of quantitative data. 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 [3, 1]. However, we followed a set of standard guidelines regarding how to perform interviews [9, 6]. For instance, we explained the purpose of the interviews, we provided clear transitions between major topics, we did not allow interviewees to get off topic, allowed interviewees to ask questions before starting the interview, and scheduled the interviews beforehand.

To structure the interviews, we composed a flexible guideline that we adjusted for future interviews depending on the answers of developers. The interviews were grounded in research questions RQ1-4 and 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 specific developers based on their background and

Table 1 General information about projects repositories.

Project	Domain	#Commits	
apache	Web Server	25,615	
bash	Interpreter	68	
bison	Parser Generator	5,423	X
cherokee	Web Server	5,748	X
dia	Diagramming Software	5,634	
flex	Lexical Analyzer	1,609	
fvwm	Window Manager	5,439	
gawk	Interpreter	1,345	X
gnuchess	Game	236	
gnuplot	Plotting Tool	8,024	X
gzip	File Compressor	445	X
irssi	IRC Client	4,130	
libpng	Image Library	2,188	X
libsoup	Web Service Library	2,005	X
libssh	Security Library	2,915	X
libxml2	XML Library	4,246	X
lighttpd	Web Server	1,470	X
linux	Operating System	445,169	
lua	Programming Language	83	X
m4	Macro Expander	953	X
mpsolve	Mathematical Software	1,434	
rcs	Revision Control System	915	X
sqlite	Database System	553	X
vim	Text Editor	5,720	

(X) Projects that we performed repository mining.

answers. This is a standard approach to cover a topic broadly. In particular, for each interviewee, we considered a subset of the following questions gathered throughout the interviews:

- **T1.** In which situations do developers use conditional directives?
- **T2.** Have developers thought about alternatives for preprocessor directives?
- **T3.** When would developers choose to use `#ifdefs` versus C-based IFs?
- **T4.** How do developers test different macro combinations in their code?
- **T5.** Do developers test all different macro combinations?
- **T6.** Do developers find that `#ifdefs` hinder code understanding?
- **T7.** Do developers find bugs in the code due to wrong `#ifdef` usage?
- **T8.** What do developers think about directives that split up parts of C constructions?
- **T9.** Do developers use tools to test the code?
- **T10.** Which types of warnings and bugs do developers check before submitting new code versions?
- **T11.** Do developers use different strategies to test code containing several conditional directives?
- **T12.** How do developers perceive the use of conditional directives inside function bodies?

In addition to these questions, we used code snippets to ask developers concrete questions about code. For each interview, we selected code snippets from that specific developer, selected

from the code repository, and sent it per email before the scheduled interview. By providing familiar code snippets, we reduced the level of abstraction in our interviews.

We performed both phone and 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. We recorded all phone interviews and created transcriptions subsequently.

To analyze the interview transcripts and emails, we again follow established research methods. We broke up the interviews into sentences and paragraphs and classified them into sets by using keywords, a process called ***coding*** in grounded theory [3]. By analyzing the keywords, we organized them hierarchically to define concepts and categories using mind maps. To connect our keywords, we started writing ***memos***, which are sentences to connect concepts and categories with the purpose of creating relationships [3]. We performed coding for each interview transcript and email and incrementally updated our memos with all new information. We met weekly to discuss the memos and noticed that interviewees progressively started to give similar answers, i.e., a situation called ***saturation*** in grounded theory [3]. At this point, we considered the topic sufficiently clear and focused on other topics that needed further elaboration. For instance, topic *T5* quickly became saturated and we removed it from the topics of future interviews. Thus, we could focus on other topics such as *T9-11* which arose during earlier interviews. The specific coding outcome is listed in Appendix 2 and 3.

To select participants for the interviews, we analyzed the 24 projects of our corpus and searched for developers that commit code containing preprocessor conditional directives. During the selection of interview participants, we needed developers who have experience in using the preprocessor. We selected developers with `#ifdef` experience because they actively use the C preprocessor in open-source development, understand the purpose of different preprocessor macros, and deal with real bugs related to preprocessor usage. In addition, such developers have more practical experience to talk about the strengths and drawbacks of the C preprocessor. For each developer that we identified as using conditional directives, we also measured code churn to identify developers that use the C preprocessor actively. For each project, we detected a group of developers that was responsible to introduce and remove the majority of conditional directives. We considered the top 10 % of developers with the highest code churns as potential interviewees. We sent emails asking those developers to participate in our study. Table 2 presents a characterization of the participants of our interviews. We sent emails to 213 developers, and 32 (15 %) participated in our interviews. By selecting only developers experienced with conditional compilation, our sampling strategy may bias the results; however, our survey eliminates this bias, as we will discuss in the next section.

In addition, we also explored whether developers from industrial projects would provide additional insights. Toward the end of our interview phase, we asked developers from industry to participate. Using convenience sampling, we sent emails to project leaders of three companies and asked them to invite developers. We used our personal contacts to identify such projects leaders. We did not use a rigorous criteria to select industry participants, and only 8 developers from Brazilian companies accepted to participate in our study.

Study 2: Survey

Whereas our interviews were designed to elicit qualitative insights into practices and reasons, surveys are designed to collect quantitative data from a large population. We designed the survey after completing and evaluating the interviews (Study 1). 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.

Table 2 Characterization of interview participants sorted by developers' ID.

ID	Experience	Projects	Media
P01	More than 5 years	*gawk*	Phone Interview
P02	More than 5 years	*bison*, *gzip*, *m4*	Phone Interview
P03	More than 5 years	*libpng*, *linux*	Phone Interview
P04	More than 5 years	*cherokee*, *linux*	Phone Interview
P05	3–5 years	*bison*	Phone Interview
P06	3–5 years	*libssh*	Phone Interview
P07	More than 5 years	*bison*, *gzip*, *m4*, *rcs*	Phone Interview
P08	More than 5 years	*linux*, *lttng*, *lttv*	Email Interview
P09	More than 5 years	*linux*, *match*	Email Interview
P10	3–5 years	*libssh*, *mig*, *sails*	Email Interview
P11	More than 5 years	*apache*, *gcc*	Email Interview
P12	1–3 Years	*sqlite*	Email Interview
P13	More than 5 years	*aap*, *vim*, *zimbu*	Email Interview
P14	More than 5 years	*sqlite*	Email Interview
P15	More than 5 years	*bison*	Email Interview
P16	More than 5 years	*linux*	Email Interview
P17	More than 5 years	*bison*, *linux*	Email Interview
P18	More than 5 years	*cherokee*	Email Interview
P19	More than 5 years	*dia*	Email Interview
P20	More than 5 years	*fvwm*, *linux*	Email Interview
P21	More than 5 years	*dia*, *libsoup*	Email Interview
P22	More than 5 years	*flex*	Email Interview
P23	More than 5 years	*gnuplot*, *linux*	Email Interview
P24	More than 5 years	*gcc*, *lighttpd*, *linux*	Email Interview
P25	3–5 years	*linux*	Email Interview
P26	More than 5 years	*libsoup*	Email Interview
P27	More than 5 years	*dia*, *libsoup*, *libxml2*	Email Interview
P28	More than 5 years	*bash*	Email Interview
P29	More than 5 years	*libsoup*, *libxml2*	Email Interview
P30	More than 5 years	*libssh*	Email Interview
P31	More than 5 years	*industry*	Email Interview
P32	More than 5 years	*industry*	Email Interview
P33	More than 5 years	*industry*	Email Interview
P34	More than 5 years	*industry*	Email Interview
P35	3–5 years	*industry*	Phone Interview
P36	3–5 years	*industry*	Phone Interview
P37	1–3 years	*industry*	Phone Interview
P38	3–5 years	*industry*	Email Interview
P39	3–5 years	*bison*	Email Interview
P40	More than 5 years	*gawk*, *sendmail*	Email Interview

With the survey, we explore topics that were unclear from the interviews or where we would like additional quantitative data (i.e., we did not ask survey questions about all interview findings). Especially if we received weak or controversial opinions from interviewees or wanted to generalize from opinions stated in interviews, we designed corresponding survey questions. For example, it did not make sense to ask our survey respondents about classes of `#ifdef` usage since we already since the majority of the developers we interviewed already provided consistent and saturated answers to that question that aligned well with findings from prior studies [5]. In contrast, we asked a more general population about their preferences toward undisciplined annotations for which we received mixed feedback in interviews.

We performed an online survey to reach more developers and again followed common guidelines for that research method [4]. We designed and refined the survey in discussions over several iterations. Specifically, we asked the following survey questions:

- **SQ1.** What is the acceptable level of conditional directives nesting?
- **SQ2.** Do developers prefer to handle portability concerns by implementing different functions or by adding conditional directives inside function bodies?
- **SQ3.** Do developers prefer to use `if` statements instead of conditional directives?
- **SQ4.** How positive or negative is the impact of using directives that split up parts of C constructions on **code understanding**?
- **SQ5.** How positive or negative is the impact of using directives that split up parts of C constructions on **code maintainability**?
- **SQ6.** How positive or negative is the impact of using directives that split up parts of C constructions on **error proneness**?
- **SQ7.** How often do developers encounter bugs related to preprocessor directives?
- **SQ8.** How easier or harder is it **to introduce** bugs related to preprocessor usage compared to other bugs?
- **SQ9.** How easier or harder is it **to detect** bugs related to preprocessor usage compared to other bugs?
- **SQ10.** How critical or uncritical are bugs related to preprocessor usage compared to other bugs?

For several questions, the survey included code snippets to make questions more concrete. The snippets are simplified examples adapted from snippets without our corpus. They are similar to those used throughout this paper. Appendix 4 lists the entire survey, depicting the questions as they were sent to developers, including those snippets.

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 collected developers information from the 24 projects in our corpus by analyzing the commit authors. From the list of all developers in those projects, we excluded developers that had already participated in our interviews and used developer names and email addresses to remove duplicates. We developed an algorithm that allows us to randomly select a specific number of developers from this list. With it, we randomly selected developers from that list and sent emails asking them to participate. We sent emails to 3,091 developers and 202 (6.5%) filled out our survey.

We use the quantitative findings from our survey to validate the findings of our interviews. We find that the results obtained from the more general population of the survey align with the results from our interviews. Our cross validation with the survey eliminates the possible bias from interviewing only experienced developers.

Table 3 Repository Mining Regarding Undisciplined Annotations.

Project	#Commits	Active Developers	Undisciplined Annotations
bison	5423	18	58
cherokee	5748	18	40
gawk	1345	3	145
gnuplot	8024	8	7827
gzip	445	7	9
libpng	2188	2	2174
libsoup	2005	101	28
libssh	2915	21	73
libxml2	4246	96	231
lighttpd	1470	2	1407
lua	83	3	2
m4	953	5	943
rcs	915	2	915
sqlite	553	13	479
Total	36313	299	14331

Study 3: Repository mining and static analysis

After the interviews, we performed additional evaluations to detect developers who heavily use undisciplined annotations. This study aimed to detect the reasons to use this type of annotation in practice, which was one of the most controversial issue in our interviews and we wanted to investigate it in more detail. We mined software repositories to analyze different versions of the source code and applied static analysis to detect undisciplined annotations. In particular, we answered the following mining questions:

- **MQ1.** What is the percentage of undisciplined annotations introduced by each developer?
- **MQ2.** What are the reasons to use undisciplined annotations?

To answer **MQ1**, we analyzed each commit of in 14 projects of our corpus (see column *Repository Mining* in Table 1). We did not use the full corpus, because we excluded projects with fewer than 10% of active developers. To detect undisciplined annotations, we used the tool *cppstats* from prior work [11], which we extended to perform the analysis on each commit (instead of on each file) of the software repositories.

Regarding **MQ2**, for each undisciplined annotation, we identified the developer who introduced it from the version control history. We sent emails to developers who introduced several undisciplined annotations, asking them about their reasons for introducing specific undisciplined annotations that we found. We sent customized emails with a couple of undisciplined annotations introduced by each developer. We selected the undisciplined annotations by searching for recurrent patterns. Previous studies [11, 12] listed a set of undisciplined annotation patterns, which we considered to select undisciplined annotations. This way, we selected the most frequent patterns in each project to ask developers additional information about.

We sent emails to 21 developers who introduced undisciplined annotations in our corpus. Four (19%) of those developers replied. Of those four, three had already participated in our interviews, but provided additional information on specific undisciplined annotations.

2 Interview Coding

After performing the interviews, we analyzed the data to identify categories, concepts, and codes. In this section, we summarize the set of codes resulting from the analysis hierarchically according to their concept. In addition, we classify the concepts into categories. In total, we have 10 concepts grouped into 4 categories.

Category 1. Strengths and Drawbacks

The key points and common problems of using the C preprocessor.

Concept 1.1. Key points

The advantages of using the C preprocessor.

1.1.1	Performance	The C preprocessor is a simple and overhead-free tool.
1.1.2	Language limitations	It solves limitations of the language.
1.1.3	Availability	No additional tool is necessary, the preprocessor is included in a wide range of C compilers.
1.1.4	Widely used	The C preprocessor is still widely used in practice.
1.1.5	Elegant solution	The C preprocessor is an elegant solution when used correctly and carefully.

Concept 1.2. Common problems

Common problems developers deal with when using the C preprocessor.

1.2.1	Number of configurations	It grows exponentially with conditional directives.
1.2.2	Mixing of languages	The C code and the preprocessor use different languages.
1.2.3	Impacts code quality	Obfuscate the code, impacting readability and maintainability negatively.
1.2.4	No usage control	It is a lexical preprocessor, and developers can encompass even single tokens with conditional directives.
1.2.5	Dead code	Sometimes no one knows that specific optional blocks of codes are not being used anymore.

Category 2. Practical use

The way developers perceive the practical use of the C preprocessor.

Concept 2.1. Conditional directives

Developers need preprocessor conditional directives because of the following reasons.

2.1.1	Portability	To handle different operating systems, platforms and libraries.
2.1.2	Language limitations	Help to solve limitations, such as header guards.
2.1.3	Optimizations	Developers do not trust compilers and need conditional directives to optimize their code.
2.1.4	Features	To include macros only when needed and select alternative macros.
2.1.5	Code changes	Developers can switch between implementation versions.

Concept 2.2. Alternatives

The alternatives to conditional directives.

2.2.1	Language constructions	Replacing conditional directives and macros with if statements, variables, and enumerators. This way, developers avoid directives and macros.
2.2.2	Design and encapsulation	Use functions, files, and directory structure to encapsulate portability concerns.

Concept 2.3. Code guidelines

Developers recommend to follow guidelines when using conditional directives.

2.3.1	Split up constructions	Do not split parts of constructions with conditional directives.
2.3.2	Directives inside function bodies	Encompass at least complete functions, or use directives only at the beginning and end of source files.
2.3.3	Complete blocks	Encompass only code with balanced brackets and complete code blocks.
2.3.4	Code clone	Use wrapper functions to avoid code clones when using different functions to handle portability concerns.
2.3.5	Compiler warnings	Avoid compiler warnings when substituting preprocessor macros with variables and enumerators.
2.3.6	Nesting	Avoid nesting conditional directives.

Category 3. Testing

The use of preprocessor conditional directives impacts testing.

Concept 3.1. Difficulties

Conditional directives increases the testing matrix.

3.1.1	Several configurations	It is unfeasible to test all configuration in real projects.
3.1.2	Platform and compilers	Developers need to consider different platforms and compilers to which they have no access.
3.1.3	Inefficient testing	There is no easy way to test everything.

Concept 3.2. Community support

End-users support developers during testing activities.

3.2.1	Testing support	End-users have different operating systems and platforms. This way, developers rely on end-users to test the code on different compilers and platforms.
3.2.2	Bug reports	The community tests the code and reports bugs.

Concept 3.3. Tool support

Developers use tools to support testing activities.

3.3.1	Static analysis	Developers use style checkers and bug detectors.
3.3.2	False positives	Static analysis should be used daily to avoid false positives.
3.3.3	Tools	Cppcheck, Coverity, Valgrind, Lint, Vera++, Coccielle, and Checkpath.

Concept 3.4. Bugs

Developers deal with bugs related to preprocessor directives.

3.4.1	Bug frequency	Bugs related to preprocessor usage happens in practice.
3.4.2	Bug types	Syntax errors, type errors, linking problems, incorrect macro expansion, incorrectly changes in control flow, and missing variables and functions.

Category 4. Refactoring

Refactoring can be applied to improve code quality.

Concept 4.1. Complexity

Refactoring becomes harder with preprocessor directives.

4.1.1	Refactoring purpose	To improve code quality mainly focusing on improving readability and maintainability.
4.1.2	Understand to refactor	Developers need to understand the code well in order to refactor.
4.1.3	Reluctant to change	Developers sometimes do not want to change a code that works.
4.1.4	Refactoring correctness	Refactoring must be correct. Developers worry about changing the machine code since it can make the code slower.

3 Key Quotations

After finishing the transcriptions of all interviews, we highlighted the must important sentences and paragraphs. In this section, for each participant, we list their important quotes. In addition, we correlate the quotes with their respective concepts and category.

Participant 1

2.1.1	Portability	Mostly, I use it for portability issues where you are dealing with different operating systems, and different facilities may or may not be available. In certain systems you may not have a specific signal, internationalization, and so on.
2.2.1	Language constructions	I prefer runtime variability instead of macros. I also define inline functions instead of function-like macros.
2.2.2	Design and encapsulation	I try to define a function that does something per system version, isolating portability concerns.
2.3.4	Code clone	Sometimes you cannot avoid it entirely, I try to minimize it.
3.1.2	Platforms and compilers	I do not have 25 different unix systems, I basically build the code on 64 bit and Linux. I try to use several different compilers and then, there are other people that build the system for me.
2.3.1	Split up constructions	It was really hard to follow. When it is overly used, it does make the code harder to read, which I try to avoid. That is why I use separate copies of functions, or they are small. That does make a big different to understand. I do not have tons of `#ifdefs`.
2.3.1	Split up constructions	It makes the code very hard to change, and harder to understand and follow.
2.3.2	Directives inside function bodies	If you do this kind of thing, not very often, it is okay.

Participant 2

2.1.1	Portability	I have to use preprocessor workarounds. I use conditional directives is to make my code portable to multiple target libraries or target operating systems.
2.3.2	Directives inside function bodies	It should be in the beginning of the file, so the rest of the file can be compiled without conditionals.
2.2.1	Language constructions	Global variables are inappropriate in library context because they are not thread safe. By using other solutions, the code does not compile. So, it does not matter the solution you come up with. The preprocessor is primarily used to overcome problems that otherwise would prevent compilation.
3.1.1	Several configurations	Every time you add one macro, you have an exponential growth to the number of testing combinations. There is no easy way to test everything, and I rely a lot on my user base to report back errors
3.2.1	Testing support	I rely a lot on integration testers. If somebody has a platform they can set up a build on that platform.
3.3.3	Tools	I also rely a lot on static analyzers. There are some pretty good tools out there, and static analyzers helped me prevent mistakes in a few cases.
3.3.3	Tools	The ideal would be static analyzers on a daily basis. If there are 200 problems and at the final of the day there are 202, these are the two new ones.
1.2.3	Impacts code quality	The C preprocessor can definitely be abused if you do not define rules on how you are going to use it. There is code that is hard to decipher. I will also say that it is a very elegant solution if there are projects rules on how it is supposed to be used.
2.3.1	Split up constructions	Because it does not divide an expression at all. I think that preprocessor directives should never break the middle of an expression, if it can all be avoided.
2.2.1	Language constructions	That is the kind of approach I like. Use local variables instead of macros. Avoid conditional compilation and only the definition depends on the preprocessor. Now the CPP is out of the way and the C code becomes clear.
2.3.1	Split up constructions	Breaking an expression with preprocessor conditional is not only risk, but if it is is a function-like macro, that is an invalid syntax since you cannot use `#ifdef` inside macro expansion.

Participant 3

2.1.4	Features	In principle, macros allow to remove optional features (you do not need).
2.1.3	Optimizations	You cannot rely on the compiler to do optimizations. The best compilers do such as GCC, but many other compilers do not.
2.2.1	Language constructions	The way we have done this kind of test is to avoid checking compilers whenever as possible. We do not check compilers any long. It becomes a messy.
2.3.3	Complete blocks	My rule is that preprocessor directives should encompass only balanced brackets.
3.1.1	Several configurations	Developers test in a very narrow way. Conditional compilation changes the code, and people usually check only one configuration. Many parts of the code do not pass by the compiler.
3.1.2	Platforms and compilers	The effort to test is too high because of the number of versions and operating systems.
2.3.1 4.1.1	Split up constructions Refactoring purpose	I try to rewrite code like that. Classic example that looks like something that has been maintained in a very incremental fashion to add new operating systems. I consciously try to rewrite things like that.
2.3.1	Split up constructions	It is very hard to maintain that code.
2.3.4	Code clone	It is dangerous because of code duplication. I do not like code duplication at all, e.g., inside switch cases.
2.2.1 2.3.5	Language constructions Compiler warnings	Replacing `#ifdefs` with if statements is a very good point. However, some compilers warn about uninitialized variables.
2.2.2	Design and encapsulation	I believe that it is a good idea to define each function in a different file and use the linker to do the work.

Participant 4

3.1.1	Several configurations	I want to check if the real behaviour works with all other real time features. If you ask whether I'm doing a combination of all, I'm not doing that. I'm reducing the scope with all macros active.
2.1.3	Optimizations	Basically most of my programming code written in C is heavily towards optimization.
2.1.3	Optimizations	I think the compiler is not smarter enough to make those decisions.
3.1.2	Different platforms and compilers	The main bugs we figured out when working on different architectures.
1.2.3 4.1.2	Impacts code quality Understand to refactor	They make code reading very hard. So, I would be against that. I would refactor the code to remove the incomplete annotations if I understand the code very well. It should have documentation explaining why they write the code like that, and why specific architectures require that.
2.3.1	Split up constructions	I would encompass the complete block of optional code with preprocessor directives.
2.3.4	Code clone	It would introduce a lot of extra work when this function changes.
2.2.2	Design and encapsulation	Your binary will get smaller.

Participant 5

3.1.2	Different platforms and compilers	Most of the time, I try with GCC and Clang with recent versions and mostly on Linux (Ubuntu or GDM). But, in some cases, I try to test with other platforms, but that is not easy.
2.3.1	Split up constructions	It is a kind of weird. I see that this kind of code is going to be hell, and that is impossible to debug.
2.3.4	Code clone	I think I would define a different function for each operating system. There is a risk of code duplication, then, I would try to avoid that as much as possible. At that point code, duplication is preferable then the obfuscation we have here.
4.1.3	Reluctant to change	One thing is to not fix what is not broken. The problem is that to refactor a code you have to understand. If you do not understand, it is not easy to refactor. Many developers would say: I am not going to touch that.
1.2.2	Mixing of languages	We do not mixing languages and the control flow.
2.3.1	Split up constructions	I do not like to split statements.

Participant 6

2.2.1	Language constructions	If it is a code that I'm writing from scratch, then I totally avoid preprocessor directives in my way as much as possible.
1.1.4	Widely used	If it is open-source code, it would have preprocessor directives everywhere.
3.1.2	Different platforms and compilers	We want to support many platforms, so, we have to test in different platforms.
1.2.3	Impacts code quality	Very incredible, dangerous, and error-prone code.
2.3.1	Split up constructions	Preprocessor directives and macros throughout the code, make code reading and debugging difficult. The heavy use of preprocessor directives and macro makes it poor code quality.
2.2.1	Language constructions	I'm pretty in support of that. Unfortunately, there are cases where there is no way to do that at runtime.
2.3.6	Nesting	It makes the code very difficult to read through and know exactly what portions are going to be compiled under what conditions.

Participant 7

2.2.1	Language constructions	I try to avoid preprocessor directives, but sometimes is better and faster.
3.1.2	Different platforms and compilers	I try to develop code that works in different platforms. For example, I try to develop code that works even for 112 bits platforms.
3.4.2	Bug types	Bugs related to variability happen, but variability bugs are only a small portion of all bugs.
3.1.2	Different platforms and compilers	We found problems related to different C standards like array in C99.
2.3.2	Split up constructions	Sure, but fixing this sort of thing is pretty low on the list of important things to do.
3.3.3	Tools	I do not use static analyzers, but I use GCC with all warnings enabled.
4.1.4	Refactoring correctness	If you change the code and the machine code is still the same, no problem.

Participant 8

3.1.1	Several configurations	We use continuous integration that attempts to build with coverage of most configurations.
1.2.3	Impacts code quality	It does hinder code understanding when `#ifdef` is within function body. The brain is not very good at understanding 2 levels of conditionals interleaved (c pre processor and c-level if).
2.3.2	Directives inside function bodies	As a general rule, we try to never, ever have `#ifdef` within function body. We can always lift out a helper function to do this, and wrap the entire helper function with the preprocessor conditional.
2.2.2	Design and encapsulation	I think in pretty much all cases, developers should lift out use of preprocessor conditionals and have different implementations of the same function.
3.1.1	Several configurations	We'd like to have a more extensive testing strategy. However, `#Ifdefs` bring a pretty much unlimited quantity of scenarios to test, and time available to work on our C code is unfortunately limited.
3.3.3	Tools	Coverity, and Cppcheck.
3.4.2	Bug types	Every compiler warning should be considered: type mismatch, resource leaks, uninitialized variables, null dereferences, and all the other nice things static analyzers can see. However, they can only see this if they are fed the code with all the possible preprocessor defines.
3.1.1	Several configurations	if preprocessor defines are well organized (e.g. one define that specifies the architecture type, on that defines the size of "long", etc.), it's fairly easy to add a case for this in the code. However, it's when the number of preprocessor defines grows that it makes it harder to test all possible combinations.

Participant 9

2.3.6	Nesting	My main problem is that there are macros 7 layers deep and I don't understand them.
2.3.2	Directives inside function bodies	We normally consider it bad style and avoid it.
1.2.5	Dead code	The most annoying thing is that they make the code unreadable. A lot of the `#ifdef` code is dead code but no one knows it. It just makes everything unreadable and annoying to look at. Since the code is so messy you get confused what's happening and that causes bugs.
3.1.1	Several configurations	We do have a ***make randconfig*** that tries to enable random #ifdef paths to see if the code can at least compile but that's about it.
3.3.3	Tools	Smatch, Sparse, GCC, Coccinelle, checkpatch.pl. checkpatch.pl insists that everyone add parenthesis around macros. Smatch has some basic checks for macro expansion bugs. Also, I have some personal tools that lists macros which execute a parameter twice.
3.4.2 3.3.2	Bug types False positives	In user space sometimes you don't have to care about resource leaks but in the kernel every bug is considered worth fixing. Those are all important, but it's a question of how reliable the tool is which generates the warnings.

Participant 10

3.2.1	Testing support	Usually because I don't have the same environment as the end-users.
3.3.3	Tools	I find static analysis tools for C are usually painful to use. Valgrind if I know that's something is wrong but can't figure out why.
3.1.3	Inefficient testing	If I use multiple environment settings, I find bugs.

Participant 11

3.1.1	Several configurations	Only via automatic tools like cppcheck
1.2.3	Impacts code quality	Sometimes it impacts understanding, for example, when it is around a lot of code and when several `#ifdef` are imbricated.
3.1.3	Inefficient testing	No, I test for my configuration only.
3.3.3	Tools	Gcc, Cppcheck, and Coccinelle.

Participant 12

2.3.2	Directives inside function bodies	It is a bad idea to use several conditions directives within function bodies.
3.1.3	Inefficient testing	I normally find bugs when running the tests with a different macro combinations.

Participant 13

3.2.2	Bug reports	Building with a variation of features, using a script. Only checks a subset. The rest depends on users reporting problems.
1.1.5	Elegant solution	Sometimes, it can also make the code clearer, when used well.
2.2.2 1.2.5	Design and encapsulation Dead code	I use it to handle differences between different systems and to support optional features. Factoring that out to functions would make the code complex and add a lot of "dead code".
2.3.2	Directives inside functions	Most common is a function inside `#ifdef` A that is used from code that is not inside `#ifdef` A. The script that builds with various combinations of features usually catches this.
3.1.3	Inefficient testing	I do not find bugs when just running the tests, but when running the tests with a different combination of features.
3.1.3	Inefficient testing	The only reason to skip it is the effort required. E.g. running all tests with all combination of features under valgrind takes an awful long time. Staring at the code often reveals potential problems, then running a test to verify just that part works well.

Participant 14

3.1.3	Inefficient testing	Normally I have a few (6-10) build configurations I'm interested in. I use continuous integration on those to run unit tests after commits.
2.3.1	Split up constructions	Not aIways. I find that the most confusing `#ifdefs` are those that change the block.
2.2.2	Design and encapsulation	For large projects I generally have an os.h with implementations for each platform meaning that all other code is platform agnostic.
3.4.2.	Bug types	Usually the problem is my particular platform falls into the incorrect default case.
3.1.3	Inefficient testing	Normally find through test failures. Try to limit number of occurrences to a handful stop easy to check correct code us running manually. Difficult with lots of third party dependencies sensitive to many flags though, e.g., Cairo, Freetype, etc.
3.3.3	Tools	I've used Vera++ to catch code that looks dodgy. Otherwise plenty of profilers. Very sleepy and valgrind deserve mention.
3.4.2	Bug types	Biggest problems have been memory/resource leaks (particularly in error cases), memory corruption, and race conditions (causing inconsistent internal state). Also concerned with potential security issues. E.g. possible buffer overflow. SQL injection. Etc. Uninitialized memory has somewhat decent warnings and tools. Null defeferences and other runtime issues are generally easy to track down due to having a stack trace.

Participant 15

3.1.1	Several configurations	All combinations.
2.2.2	Design and encapsulation	The alternative would be to have many similar functions, which could make the code difficult to understand too.
3.4.2	Bug types	In my code I do my best to remove ALL bugs, of any kind.

Participant 16

3.1.3	Inefficient testing	I just use whatever I need and stop when things work.
2.1.4	Features	If they are conditional directives used sparingly they actually enhance my understanding by showing that certain code blocks are only required for certain kernel configurations.
2.1.4	Features	I think we could do without but then the clarity that certain pieces of code are not really used for common cases would be lost.
3.4.2	Bug types	The typical kernel bugs. In my subsystems debugging options are often in `#ifdef` statements which may sometimes hide bugs.
3.4.2	Bug types	All bugs and all new warnings need to be checked.

Participant 17

2.1.1	Portability	To be able to build the software on different platforms and operating systems.
3.3.3	Tools	Using a validation program that tries many combinations.
2.3.3	Complete blocks	It's not easy to see if the brackets are balanced, or to know if using the "%" key will correctly find the corresponding closing or opening bracket.

Participant 18

2.2.1	Language constructions	Mostly by policy, I strictly follow zero-configuration in my projects. Additionally it makes the makefiles VERY hard to read, this is especially annoying when maintaining external projects.
3.1.3	Inefficient testing	I only check the pre-selected default combination if possible, following my zero-configuraiton policy.
1.2.3	Imapcts code quality	The criticism is well deserved. However, when used correctly it is not a problem.
2.2.2	Design and encapsulation	I personally like to separate code on the directory/file structure level. Eg. OS specific files are in a relevantly names OS directory and only compiled when building for the given OS. This way no preprocessing is necessary as it is taken care of on the makefile level, additionally it is very portable and requires no special tools.
1.2.3	Impacts code quality	It also makes the makefiles nearly unreadable.
1.2.3	Impacts code quality	It makes the code a maintenance nightmare as it takes way too much time to read it. Luckily my text editor is able to use syntax highlighting depending on my pre-selected definitions. However it still takes time to check that all the combinations result in correct code.
2.3.1	Split up constructions	I prefer b because it does not break the if statement in the middle, making it more readable.

Participant 19

2.1.1 2.1.4	Portability Features	In libraries which implement portability abstractions, such as GLib. To support optional features
3.1.1	Several configurations	It multiplies the testing matrix.
3.1.3	Inefficient testing	Compile multiple times.
2.3.1	Split up constructions	I prefer `#ifdef` blocks that act as complete statements.

Participant 20

2.1.1 2.1.2	Portability Language limitations	I use it mainly due to limitations in the language or compilers: as header guards, to fix portability issues, to make simple compatibility layers, and on code that has a high chance of containing errors due to all the repetitions.
1.2.3	Impacts code quality	The C preprocessor is a necessary evil, much like the goto statement.
1.1.6	No additional tool	Introducing another tool as a dependency to a project has drawbacks and the preprocessor is always there for C/C++.
2.2.1	Language constructions	When C-based IFs can't be used or when the code is error prone due to the repetitive nature.
1.2.3	Impacts code quality	The preprocessor can make the code easier or harder to understand depending on how they're used.
2.3.1	Split up constructions	I avoid these kinds of directives. They make the code hard to understand and maintain. It is very easy to make mistakes and very hard to find them.
2.3.4	Code clone	Repetition of code often leads to bugs in the future.
2.3.2	Directives inside function bodies	Logic is not interrupted. So, it is usually easier to understand.
2.3.1	Split up constructions	My gut feeling keeps screaming possible bugs when I'm faced with a code like that. Can't confirm unless I carefully check the rest of the code, and even them I might miss them. In one word: BAD

Participant 21

3.1.3	Inefficient testing	Usually checking is straightforward since I try to avoid nested `#ifdef`.
2.3.1	Split up constructions	It makes the code too complex, and might lead to bug(s).

Participant 22

2.1.1	Portability	To handle different configuration, inlining code, and a quick and dirty aspect oriented programming.
3.1.3	Inefficient testing	I try out macro combinations by hand.
2.1.3	Optimizations	Optimization when I know the if'ed code will never be reached (configuration) - cross-platform includes (an if would not even compile I suppose)
1.2.3	Impacts code quality	That is a fear for code that I need to audit (code written by others). You never know what can happen in a project that makes a big usage of preprocessor macros.
2.3.1	Split up constructions	It is horrible to read and maintain. It should be used only for good reasons and probably be a lot more documented. Orphan curly braces in `#ifdefs` should be avoided if possible they are a nightmare to debug.
2.2.2 2.3.4	Design and encapsulation Code clone	I would prefer code with more platform specific functions and `#ifdefs` around these function calls but then again code should not be duplicated.

Participant 23

2.2.1	Language constructions	If something can reasonably be done without the preprocessor, I choose that way. It's much more flexible once the binary is there to enable functions at runtime or with a configuration file than having to recompile the project again.
3.1.1	Several configurations	I only check on the supported configurations, using automatic compilation and test scripts. If the program runs elsewhere, that's a bonus, but not a target.
1.2.3	Impacts code quality	`#Ifdef` code can become a real mess and grow without any control
2.3.6	Nesting	I use an IDE that can grey out dead parts of the code, it helps but it's not always perfect. I have seen parts of code with so many nested `#ifdef` statements that I wonder how it even compiles on some architectures. if your function needs a nest of many intricate `#ifdefs`, you probably should write different versions of that function to avoid that situation
3.4.1	Bug frequency	Bug happens a lot. Most of the time the developer is to blame.
2.3.2	Directives inside function bodies	Most of the time it happens when some code uses more than one `#ifdef` statement inside a function body.

Participant 24

1.2.3	Impacts code quality	Make the code messy. Complicate static analysis and compilation. They generally indicate that I'm working around a problem rather than fixing the underlying software, e.g. by defining a stable API at a lower level.
3.1.3	Inefficient testing	Generally check only the platform I use daily, with all optional features enabled. This means my code is often broken on the other `#ifdef` paths.
2.1.4	Features	Not making features optional. Instead, just making a decision over whether the project should have the feature.
2.3.1	Split up constructions	Incomplete directives are hard to read.
3.4.2	Bug types	Compilation failures due to syntax errors, linking problems, etc.
2.3.1	Split up constructions	I find it very hard to see the actual control flow which would result. This code is insane.
2.2.2	Design and encapsulation	If you really have to support that many platforms (and nobody does), you should really use three different versions of the entire function body, and put any common prologue and epilogue code in a wrapper function.
2.3.1	Split up constructions	Because the `#ifdefs` are at the statement level, rather than the expression level.

Participant 25

3.1.1	Several configurations	I usually check macro combinations via builds with multiple configurations. I do not usually check all possible macro combinations.
2.3.1	Split up constructions	I think encompassing only parts of statements is a bad idea and hinders code readability. I try to avoid doing this, and I am under the impression that others often try to avoid this as well (e.g. the Linux kernel coding guidelines are even stricter and suggest that using `#ifdefs` within functions at all is a bad idea. I'm not sure I would go quite that far, but the code above seems bad).
2.3.1	Split up constructions	I find this easier to read and like to avoid partial language constructs being enclosed in `#if` blocks.

Participant 26

3.1.3	Inefficient testing	I check whatever combinations I can. Some of the combinations can only be tested on systems to which I have no access, in which case I rely on others to help out, or just cross my fingers.
2.3.1	Split up constructions	Breaking the nested structure of C blocks with preprocessor directives should be avoided if possible because it is harder to read and understand.
2.2.1	Language constructions	I prefer runtime that avoids `#ifdefs`.

Participant 27

2.1.1 2.1.4 2.1.5	Portability Features Code changes	Conditional features that can be optionally included or excluded. Conditional code to deal with differences between systems and libraries. Temporarily commenting out sections of code (`#if 0`), often as bugs are fixed and the old code is left in place before being removed
3.1.3	Inefficient testing	I usually don't check all possible macro combinations.
1.1.6	No additional tool	The C Preprocessor is a tool like any other. Its use requires understanding its limitations and what you want to get out of using it.
3.1.3	Inefficient testing	I do occasionally find problems due to insufficient testing of infrequently-used features that can be conditionally included
2.3.1	Split up constructions	I think it could be cleaner. It's not exactly my style, but if it's used consistently and the developers are familiar with it, it should not be a problem.

Participant 28

2.1.5	Code changes	To comment a block of code, also I keep existing preprocessor directive as of now when preforming change to an existing code base.
2.3.1 1.1.5	Split up constructions Elegant solution	I do not feel it always make the code hard to read and understand. Usually it encloses a set of platform specific includes (Linux kernel). As a hammer version of comment a block of code (though only in devel mode) it is fine too. As a mean to select a block of code I find it puzzling when many level of imbrication exists and the code enclosed spans pages (WebKit comes to mind).
2.3.2	Directives inside function bodies	For no good reason, except I like this way to avoid the mental context switch from c to cpp.

Participant 29

2.3.4 2.3.1 2.3.5	Code clone Split up constructions Compiler warnings	I accepted this incomplete annotation because not doing it that way would require duplicating code. That's the sort of code that might trigger "condition is always true" warnings in compilers/analysis tools. I don't like that either.
1.2.3	Impacts code quality	In this case, given that the whole thing encompasses only 11 lines of very simple code, I think it's pretty clear.
4.1.3	Reluctant to change	If I had some tool that didn't deal well with that code, then I'd probably rewrite the code. But as far as I know, I don't.

Participant 30

2.1.1 2.1.4	Portability Features	I need to handle different operating systems and select implementation alternatives.
2.3.1	Split up constructions	I can understand that the form is a little weird. When writing that code, I probably thought that I didn't want an empty brackets pair. If I needed to rewrite that code today, I'd write differently.
2.3.1	Split up constructions	In this particular snippet, the danger is very slim, because the whole problematic code only makes 8 lines and this anti-pattern doesn't repeat everywhere else. Otherwise, I agree that it could cause problem in bigger sections of code.
2.3.1	Split up constructions	I understand guidelines are important for the homogeneity of a big project like Linux. In fact we often ask contributors to rewrite patches to follow better our conventions. Other than that they're often very personal preferences like the code formatting.
2.3.1	Split up constructions	The code was actively rewritten at the time and it often happens that first drafts of an idea ends up in poor code. At least I rewrote that one very quickly.

Participant 31

Translated from Portuguese (PT-BR).

3.3.3	Tool	Coverity, Splint and Valgrind.
3.1.1	Several configurations	The features were built and scaled up over time. Thus, when a feature was ready, it would be enabled. That is, we test the functionality completed with the continuous integration system.
3.1.1	Several configurations	We test on all platforms with the maximum feasible settings to that platform.
2.3.1	Split up constructions	It was something to be avoided but, if necessary, was accepted but with many caveats and demanding different justifications. In summary, I would say it was not accepted.

Participant 32

Translated from Portuguese (PT-BR).

3.1.1	Several configurations	It is terrible to use `#ifdef`, which should be reserved for mutually exclusive options, A, B and C are different platforms. In this case, we try to obtain 100% coverage.
3.1.1	Several configurations	I test on all platforms considering the different code configurations. However, we use `#ifdef` exclusively to support different platforms, and it gives a compile error if they were not excluded from compilation in other platforms. Any use different creates a combinatorial explosion of execution paths.

Participant 33

Translated from Portuguese (PT-BR).

3.1.1	Several configurations	Depends on the project. Generally when using `#ifdef` to functionality, they are conflicting features. If so they can be used together, they are tested together.
2.3.1	Split up constructions	We usually avoid this type of annotations, but there are projects where it is allowed (depends on who is the project owner).
3.3.1	Static analysis	You send the code for review, and the review process performs the analysis, and sometimes tools are executed to check the code. But in most cases, we use static analysis tools and coding style checkers.
3.3.3	Tools	Cppcheck, Lint and CODAN.

Participant 34

Translated from Portuguese (PT-BR).

3.3.3	Tools	The code's style was defined by the company. All functions, variables, etc., had to follow the pattern of the company. They even made a simple tool that all checked the code before submission.
3.1.1	Several configurations	We test many configurations. We also use preprocessor macros TEST_MODE and DEBUG_MODE.

Participant 35

Translated from Portuguese (PT-BR).

3.1.1	Several configurations	I work in different industry projects. Normally we have to test all supported platforms and configurations.

Participant 36

Translated from Portuguese (PT-BR).

2.1.3	Optimizations	Preprocessor macros can be used to remove the most tedious and error prone parts of programming. It's also the only C-native way to conditionally compile when run-time checks are unnaccepable to performance. There are no alternatives to the C preprocessor for this type of usage without using some tool outside the language.
3.1.1	Several configurations	We normally test all supported platforms and configurations.

Participant 37

Translated from Portuguese (PT-BR).

2.2.1	Language constructions	I would really like to see something like pattern matching in the C preprocessor. So capturing statements inside an expression of macro arguments would be cool. Often static inline functions are very sufficient and replace macros most of the time. Also, another example is to emulate something like static variables.
2.2.1	Language constructions	One can embed m4 into the gcc toolchain easily, I played around with that but because it seems not acceptable for use in upstream projects and I don't have much experience with it, I didn't follow up on this.

Participant 38

Translated from Portuguese (PT-BR).

3.1.1	Several configurations	It is important to check different platforms. We do not support many.
2.2.2	Design and encapsulation	The way that the C pre-processor is used, as with most other programming language features, is highly dependent on the context it is used in. For example, the question about whether `#ifdef` should be used to separate OS specific functions/code, or whether the code should be in a different file is really dependent on how much OS specific code there is. If only one extra function is required, then using `#ifdef` might be preferred, if there are several functions then separating into different files is probably preferable.
2.2.2	Design and encapsulation	The idea is to put the messy preprocessor checks in header files and keep the main body of code clean.

Participant 39

4.1.3	Reluctant to change	It is good to fix it. However, it has very low priority.
3.1.3	Inefficient testing	I do not test different macro combinations.
2.1.3	Optimizations	Sure, let's go to the Linux kernel for some good examples. In the fast paths, we would really like to be pushing millions of I/O operations per second. We would never scale to that level if we threw in lots of debug checks to verify assumptions, but it is important we have a way to easily enable debugging checks when we are verifying code correctness. For example, you might want to prove that all callers into a particular function are in a context that allows rescheduling, so we have a macro function called "might_resched()". If we are debugging these conditions, we turn on that debug flag and the macro is defined to verify conditions are true, taking up costly CPU cycles. If the flag is not turned on, the macro is defined to a NOOP, and so no CPU cycles are wasted.

Participant 40

2.3.1	Split up constructions	It is better because it does not divide any expression or statement.
2.1.1	Portability	Because that all was necessary, and in this case the reason for the different code is quite clear to someone familiar with the old varags mechanism and the new stdargs one.
2.3.4 2.3.1	Code clone Split up constructions	It does not impact understanding in this kind of small case. I saw no reason to duplicate code.
1.2.3	Impacts code quality	The C preprocessor can be a good solution by following guidelines.

4 Survey

This section contains the survey described in the exact form that it was sent to developers.

This survey consists of five parts:

1. General use of preprocessor directives (3 questions)
2. Use of directives that split up parts of C statements and expressions (3 questions)
3. Bugs related to the use of preprocessor directives (4 questions)
4. Background (2 questions)

You should be able to answer our survey in around 15-20 minutes. We will use your answers to understand the practical use of preprocessor directives and develop supporting tools. We really appreciate your help. Thanks!

1. Preprocessor Directives

Please express your opinion regarding the following implementation styles.

```
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.. */
}

/* Only OS1.c or OS2.c
is included depending
on the platform. */
```

(B)

Which implementation style do you prefer?

- ☐ I strongly prefer (A)
- ☐ I prefer (A)
- ☐ I does not matter
- ☐ I prefer (B)
- ☐ I strongly prefer (B)

```
#ifdef DEL
in_char("delete");

#ifdef VERIFY
printcmd();
#endif

putchar('\n');
flush();
#endif
```

Nesting level 2.

```
#ifdef USE_USG_PTYS
#ifdef USE_ISPTS_FLAG
    if (IsPts) {
#endif
#ifdef TTYSIZE_STRUCT
    // lines of code
#ifdef OPT_TEK4014
      if (TEK4014_ACTIVE(xw)){
        // lines of code
      }
#endif
#endif
// Code here
#ifdef USE_ISPTS_FLAG
    }
#endif
#endif
```

Nesting level 3.

```
#ifdef NOCWDRC
  // lines of code
#else
  if (plotrc == NULL) {
    if (tmp_home) {
#if defined(ATARI) || defined(MTOS)
#ifdef VERBOSE
        char *ini_ptr = getenv("PATH");
#ifdef FLAG
        // lines of code
#endif
#endif
#endif
    }
  }
#endif
```

Nesting level 4.

Which nesting level is acceptable?

- ☐ No nesting
- ☐ Up to 2
- ☐ Up to 3
- ☐ Up to 4
- ☐ Up to 5 or higher

```
if (*Y_AXIS.label.text) {
#ifdef PM3D
    if (rot_x <= 90){
        double step = (other_end - yaxis_x);
        // several lines of code..
        if (map)
            *t = text_angle;
    }
#endif
    // several lines of code..
}
```

(A)

OR

```
int PM3D_RT = 0;
#ifdef PM3D
    PM3D_RT = 1;
#endif
if (*Y_AXIS.label.text) {
    if (PM3D_RT && rot_x <= 90){
        double step = (other_end - yaxis_x);
        // several lines of code..
        if (map)
            *t = text_angle;
    }
    // several lines of code..
}
```

(B)

Which code snippet do you prefer?

- ☐ I strongly prefer (A)
- ☐ I prefer (A)
- ☐ I does not matter
- ☐ I prefer (B)
- ☐ I strongly prefer (B)

2. Splitting up parts of C Syntatical Units with Preprocessor Directives

Developers sometimes use preprocessor directives that split up parts of C statements and expressions as presented next.

```
while (bit_depth < 8
#if defined (PNG)
&& row != NULL
#endif
){
  // lines of code..
}
```

Developers split up a while condition with directives.

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

Developers separate an opening and its respective closing bracket with directives.

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

Developers split up a function call with directives.

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

Developers split up a function definition with directives.

How negative or positive is the impact of using directives that split up parts of C statements and expressions on *code understanding?*

- ☐ Totally negative
- ☐ Negative
- ☐ Neither negative or positive
- ☐ Positive
- ☐ Totally positive

How negative or positive is the impact of using directives that split up parts of C statements and expressions on *code maintainability?*

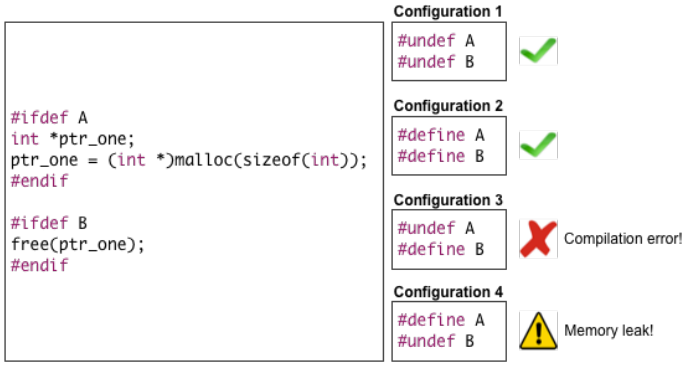
- ☐ Totally negative
- ☐ Negative
- ☐ Neither negative or positive
- ☐ Positive
- ☐ Totally positive

How negative or positive is the impact of using directives that split up parts of C statements and expressions on *error proneness?*

- ☐ Totally negative
- ☐ Negative
- ☐ Neither negative or positive
- ☐ Positive
- ☐ Totally positive

3. Bugs

Some bugs appear only when developers define a specific set of preprocessor macros, i.e., bugs that appear only in specific configurations. The next code snippet presents an example.



How often do you encounter bugs that appear only in specific configurations?

- ☐ Very often
- ☐ Often
- ☐ Sometimes
- ☐ Ocasionally
- ☐ Rarely

How easier or harder is it to introduce a bug like that compared to bugs that appear in all configurations?

- ☐ Much easier
- ☐ Easier
- ☐ Neither easier or harder
- ☐ Harder
- ☐ Much harder

How easier or harder is it to detect a bug like that compared to bugs that appear in all configurations?

- ☐ Much easier
- ☐ Easier
- ☐ Neither easier or harder
- ☐ Harder
- ☐ Much harder

How critical or uncritical are bugs that appear only in specific configurations?

- ☐ Very uncritical
- ☐ Uncritical
- ☐ Normal
- ☐ Critical
- ☐ Very critical

4. Background

Please answer the following two questions about your experience.

For how long have you been working / have worked with preprocessor directives such as `#ifdef`, `#else` and `#endif`?

- ☐ Less than a year
- ☐ 1–3 years
- ☐ 3–5 years
- ☐ More than five years

Have you worked in industry and open source projects?

- ☐ Only open source projects
- ☐ Mainly open source but also industry projects
- ☐ Industry and open source projects
- ☐ Mainly industry but also open source projects
- ☐ Only industry projects

Please use the text box to write any additional comments. If you want we can send you the results of our survey. In this case, please leave your email address.

References

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

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

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

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

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

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

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

8 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.

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

10 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.

11 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.

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

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