WHITE
PAPER

MISRA C:2012

Since its launch in 1998, MISRA C has become established as the most widely used set of coding guidelines for the C language throughout the world. Originally developed within the automotive industry, it has now been embraced within many other industries where development of robust software is a critical imperative, either for safety or commercial reasons.
This paper discusses the thinking and motivation which has led to the publication of the latest version, MISRA C:2012.

PAUL BURDEN
Member of MISRA C Working Group and co-author of MISRA C:2012

PRQA Programming Research
www.programmingresearch.com

INTRODUCTION

The MISRA C Guidelines first appeared in 1998 as an initiative within the UK automotive industry. The publication was a response to the challenges imposed by the rapidly increasing critical reliance on software in motor vehicles. MISRA C is now used across many industries and has become the most widely adopted coding standard for the C language world-wide. MISRA C:1998 (MC1) was revised with an updated version MISRA C:2004 (MC2) and this is now being superseded by a third version, MISRA C:2012 (MC3), published March 2013.

Companies who have invested heavily over a number of years in developing software in compliance with previous versions, as well as companies that are new to MISRA will be asking:

a. Is a new set of coding rules really necessary?
b. Are the new coding rules significantly better than the old?
c. Will my MC1/MC2 compliant legacy code be compliant with the new standard?

This paper provides a brief overview of MC3 and addresses these important questions. It discusses the motivation behind the project and the benefits and risks to be incurred in complying with the new standard.

THE C LANGUAGE HAS EVOLVED

Previous versions of MISRA C have specified that code should conform to the first standardized version of the C language ("ISO/IEC 9899:1990, Programming Languages – C") – commonly referred to as C90. This conservative policy has continued for a long time despite the arrival of C99 (in 1999) and C11 (in 2011).

Adherence to C90 has been sustained for at least 2 important reasons:

a. Compiler vendors, especially in the embedded C industry, have been slow to implement some of the new language features. C90 is well supported by compilers and tools, but support for the full range of features added in C99 and C11 is distinctly variable.
b. There has been a widespread view that while some attractive new features have been introduced in the C language, not all have been an unqualified improvement, particularly when reliability is a major concern. Some have undoubtedly filled important gaps in the original C language; the design of others has been questionable because they have introduced additional vulnerabilities into the language. The language standards provide appendices which list features of the language which result in undefined behavior. In the C99 standard that list is roughly twice as large as the list in the C90 standard.

It is an unfortunate feature of the standardization process, that while it is quite easy to add new features to a language, it is often impossible to 'remove' existing language problems and dangers without violating the principle that the behavior of existing source code must not change. The C language, for all its merits, has many weaknesses which will always remain, and one of the primary purposes of coding rules is to avoid those problems by defining a subset of the language which can be enforced and used with greater confidence.

The C language shows no signs of relinquishing its popularity and continues to be heavily used in safety critical software development where reliability is a prime concern. Despite reservations about the wisdom of certain developments in C99, it was decided at the outset of the MC3 project, that MISRA C should no longer be constrained to conform to the C90 version of the C language. MC3 recognizes that features such as inline functions and type _Bool are valuable additions to the language, but it also introduces 11 new rules which restrict usage of some specific C99 language features.

IMPROVEMENTS

Rule Definition

The task of defining coding rules in language that is clear and unambiguous is often far more difficult than might first appear. Considerable effort has been expended in MISRA C3 to improve the definition of existing rules in areas where divergent interpretations have emerged. The definition of a coding rule now includes sections entitled:

- Amplification – an expanded explanation of the rule requirements
- Rationale – an explanation of why the rule is necessary
- Exceptions – specific situations where the requirements of a rule do not apply
- Examples – compliant and non-compliant code examples

Jargon

When describing technical aspects of the C language, it is essential to use terminology which has a well defined meaning and is consistent with the ISO C standard. However, in defining coding rules it can be necessary to introduce some additional jargon. The “type” system in the C language presents particular problems because it contains a number of inconsistencies and anomalies. The type of an arithmetic expression as defined by the language is often an unintuitive and inadequate way of describing its true nature and so **MC2** introduced the concepts of ***underlying type*** and ***complex expression***. Unfortunately, the definition of these terms was rather loose and the concepts extended only to signed and unsigned integer types. The choice of these terms was also rather unfortunate because the term “underlying type” is used in a different sense in the C++ language definition and the term “complex” is commonly used to refer to “complex arithmetic” – an algebra which is supported in C99.

In MISRA C3, the term ***underlying type*** has been superseded by the term ***essential type***. The term ***complex expression*** is now superseded by the term ***composite expression***. Essential type provides a way of describing the type of any arithmetic expression in a way which is much more intuitive and helpful when defining coding rules. The definition of essential type in MC3 now encompasses the complete range of integer types. The type of an arithmetic expression is now characterized as being either:

- essentially Boolean
- essentially character
- essentially enum
- essentially signed
- essentially unsigned
- essentially floating

Mandatory rules

Previous versions of MISRA C have divided rules into 2 categories: the Required Rules and the Advisory Rules. Code which is claimed to comply with MISRA C must comply with every Required Rule unless the non-compliance is supported with a formal deviation. Advisory rules may be applied subject to a degree of flexibility and formal deviations are optional.

Any attempt to classify coding rules according to their perceived level of “importance” is bound to reflect some degree of subjective judgment. A number of factors have to be considered:

a. How likely is it that a violation of the rule will result in a software fault?
b. How frequently is a violation of the rule likely to be encountered?
c. Does the rule impose irksome restrictions on code which is safe?

However there are a few coding rules which are considered to be so straightforward and so uncontroversial that no circumstances are envisaged under which it should ever be necessary to raise a deviation.
In MISRA C3 such rules are classified as ***Mandatory Rules***. These are rules for which a deviation is not permitted – under any circumstances.

Enforceability

MISRA C has always emphasized the key role played by static analysis tools in the enforcement of coding rules. It has long been recognized that automatic enforcement is vital because:

a. It removes uncertainty
b. It saves time
c. It provides immediate feedback at the time when code is being developed
d. It alleviates the embarrassment and confrontation associated with manual code reviews

In practice static code analysis can do far more than simply enforce coding rules. Coding rules exist to define a safer subset of the language which will reduce the potential for bugs to be introduced. Static analysis tools are able to enforce coding rules but they also serve to identify coding errors which are often not addressed specifically by coding rules.

In most coding standards there are rules which are sometimes described as being "not statically enforceable". Of the 142 rules in MISRA C2, there are approximately 9 which can be so described. In order to comply with such rules it may be necessary to comply with requirements which cannot be determined directly or reliably from the source code; for example:

a. Documentation
b. Process requirements
c. Requirements which are loosely defined
d. Subjective interpretation of comments
e. An understanding of functional requirements

So, for example, Rule 12.10 "The comma operator shall not be used." can be enforced with complete certainty by simple analysis of the source code. On the other hand, consider the following:

a. Rule 2.4: Sections of code should not be "commented out"
b. Rule 3.2: The character set and the corresponding encoding shall be documented
c. Rule 18.3: An area of memory shall not be reused for unrelated purposes

Compliance with rules like these cannot rely on automatic enforcement because analysis of the source code is either not relevant or not sufficient (even for Rule 2.4).

MISRA C3 draws a distinction between what it terms as "**rules**" and what it terms as "**directives**". The distinction is described as follows.

> *"A **directive** is a guideline for which it is not possible to provide the full description necessary to perform a check for compliance. Additional information, such as might be provided in design documents or requirements specifications, is required in order to be able to perform the check. Static analysis tools may be able to assist in checking compliance with directives but tools may place widely different interpretations on what constitutes a non-compliance."*
>
> *"A rule is a guideline for which a complete description of the requirement has been provided. It should be possible to check that source code complies with a rule without needing any other information. In particular, static analysis tools should be capable of checking compliance with rules… "*

Scope and Compliance

The ease with which different coding rules can be enforced varies considerably. The simplest rules (e.g. "Octal constants shall not be used") can be enforced by inspection of the syntax of a single statement. Other rules require analysis of the block of code controlled by a control statement, a complete function, a complete translation unit or even the entire code base of a project.

Every rule in MISRA C3 is classified as either a "System Rule" or a "Single Translation Unit Rule", in order to reflect the scope of the analysis required to verify compliance. The distinction is significant for 2 reasons:

a. It is not possible to claim compliance with a System Rule as a property of a single translation unit. Compliance with System Rules can only be established as a property of an entire system.

b. The task of ensuring compliance with System Rules is often significantly more demanding and time consuming.

Decidability

A further important characteristic of any coding rule is an attribute known as ***decidability***. It is well recognized that the capability of coding rule enforcement tools varies widely. As noted above, some rules can be enforced easily because the level of analysis is simple – possibly requiring only a superficial analysis at the level of syntax. Enforcement of other rules may require a much deeper analysis of code structure and semantics.

However, it also needs to be understood that there are some coding rules which are intrinsically "***undecidable***" – in the sense that compliance can never be guaranteed however sophisticated the static tool analysis may be. A rule is deemed to be "***decidable***" if it is theoretically possible for a tool to identify any non-compliance in any possible situation without generating "false-positives". In other words it will be possible for a tool to supply one of 2 possible answers in any situation:

1. Yes – a rule violation definitely occurs at this point
2. No – a rule violation definitely does not occur at this point

All rules in MISRA C3 are now specifically classified as either "***decidable***" or "***undecidable***". Out of 143 rules, 28 are classified as "undecidable". It is never possible to guarantee compliance when a rule is "undecidable". When a rule is "undecidable", there will be some situations where the best diagnosis that can be obtained from a tool is of the form:

3. It is possible that a rule violation occurs but impossible to be sure.

In practice, tools will vary in their ability to diagnose a violation (or the possibility of a violation) effectively and accurately in all situations – even when a rule is classified as decidable.

MIGRATION

What will be the impact of MC3 on code which has been developed to comply with MC2 ? Although MC3 is intended to address new issues associated with C99, it remains just as relevant to code developed for a C90 environment and the additional requirements introduced in MC3 are likely to have limited impact on MC2 compliant code.

A coding rule can easily be discredited if it imposes restrictions which

- do not directly address a known software fault
- encourage code changes which are just as dangerous as the problem which the rule purports to avoid
- place too many restrictions on code which is perfectly safe

Improvements in MC3 have actually resulted in some coding restrictions being lifted. Some rules have been redrafted and a few have been removed altogether.

CONCLUSIONS

MISRA C3 is a significantly larger document than MISRA C2, but in fact the total number of rules has not increased greatly. MISRA C2 contained 142 rules. MISRA C3 contains 143 rules and 16 directives. New rules have been introduced, a number have been redrafted, and a few have been removed altogether. The document has grown in size more as a result of improvements than as a result of additional requirements. Key improvements include:

- Improved rule descriptions – including amplification, rationale, exceptions and examples
- Distinction between rules and directives
- Classification of Mandatory rules
- Distinction between system wide enforcement and single translation unit enforcement
- Recognition of the decidability issue

The impact of the new standard on code which complies with MC2 is likely to be limited.

It is well recognized that the composition of coding standards is a subject which can raise strong feelings and controversy. MISRA C has not been immune but has nevertheless garnered a widespread following because it has remained focused on the goal of establishing a safe subset of the C language which is amenable to automatic enforcement. It is significant that the proportion of rules which are classified as decidable in MC3 is very high and that support for enforcement of MISRA C rules has become a standard requirement in compilers and static analysis tools.

ABOUT THE AUTHOR

Paul Burden is a technical consultant working for PRQA. He has worked with clients around the world providing training and advice particularly in the area of coding standards enforcement. In his role as product manager for QA·C over a number of years he has gained considerable experience relating to the benefits and pitfalls of static analysis tools.

He has been a prominent member of the MISRA-C Working Group since its formation more than 10 years ago.