MISRA C:2012 Cure or Curse

Paper presented at the Advanced Engineering Conference

November 2014

First Edition
by
Eur Ing Chris Hills BSc (Hons),
C. Eng., MIET, MBCS, FRGS, FRSA

PhaedruS SystemS

The Art in Embedded Systems comes through Engineering discipline.

MISRA-C:2012 Curse or Cure?

MISRA-C:2012

Third MISRA-C

First Edition 1998

Over 16 years MISRA-C has become
The worlds most widely used
C coding standard

Is it a curse or a cure?

PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

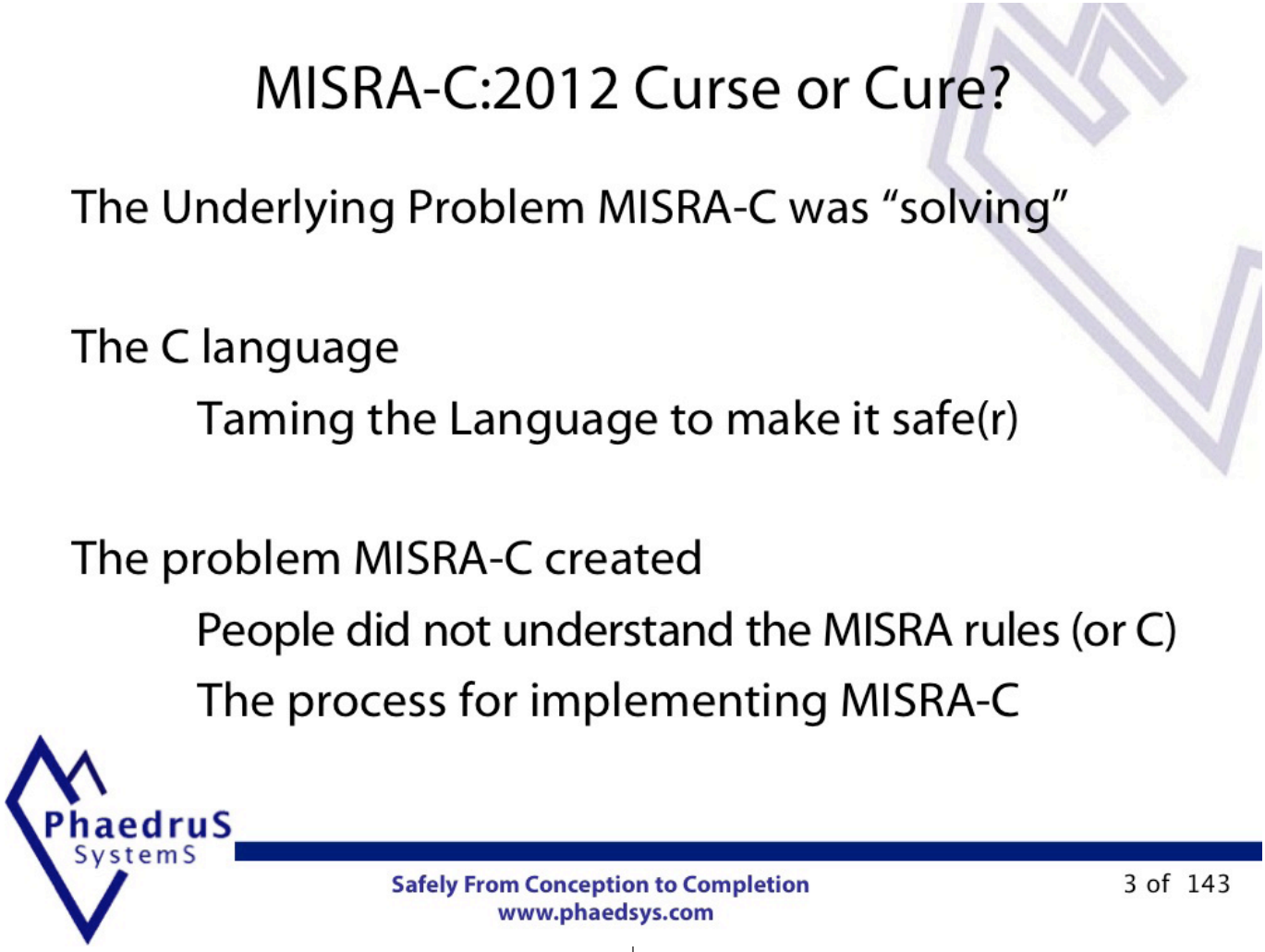
2 of 143

MISRA-C started in 1996 as part of the MISRA series of reports on automotive software development. MISRA-C was originally published in 1998 as *Guidelines for the Use of the C Language in Vehicle Based Software* and aimed at the UK automotive market. However, Programming Research, LDRA and Chris Hills, CTO of Phaedrus Systems, pushed the document to a wider audience. It soon found a home across the whole embedded, real time and critical systems markets, not just in the UK but globally. This meant that the second version of MISRA-C, published in 2004, changed its title to *Guidelines for the Use of the C Language in Critical Systems*.

Since 2004 the majority of the MISRA-C working group have come from the defence and aerospace industries with automotive representation being a minority

Over the 16 years since its first appearance MISRA-C has become the world's most widely used C coding standard. Either as straight MISRA-C or when used as the basis for company coding standards where formal MISRA-C compliance is not required, MISRA-C is in use from Japan, heading west, all the way to San Francisco.

But whilst many promote MISRA-C some call it MISERY-C. Is MISRA-C a curse or cure?



MISRA-C is there to solve a problem. But it appears it has also created a problem.

What is this problem it is trying to solve? Basically, the C language needs taming.

C, as originally devised, has an ethos of "trust the programmer". That means that, unlike many other programming languages, there no were built-in safeguards.

C is famously NOT strongly typed.

It is quite legal to store a double (say 32 bits) in a char (probably 8 bits) and there is no warning required of the loss of 3 bytes of data.

It is quite legitimate (syntactically) to increment a pointer way past the end of an array.

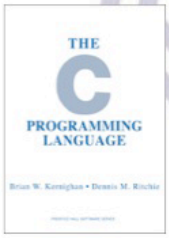
Not to mention the ever expanding list of things that are flagged by the standard as "undefined", "implementation defined" and "unspecified".

The problem is further exacerbated by the undeniable fact that most software people do not understand C. Certainly, after 17 years of involvement with C standardisation at MISRA, BSI and ISO levels, I do not know anyone who fully comprehends the whole language and its use where many things are implementation defined.

This sometimes means that a user may not understand the subtleties of the problem that a particular MISRA rule is intended to address and so do not see what the rule is trying to achieve. This in turn can lead to inappropriate application of MISRA-C which in some cases can cause more problems than the guidelines solve.

MISRA-C:2012 Curse or Cure?

- The C Language
  - Which C?
- K&R C (1978) 228 Pages



- ANSI-C (K&R2 1989)
- ISO-C 9899:1990 (C90) 271 pages
- ISO-C 9899:1990 +A1 & TC's 1-3 (C95)
- ISO-C 9899:1999 (C99) 568 Pages
- ISO-C 9899:2011 (C11) 853 Pages

Often the first question that few programmers can answer correctly is, "Which C are you using?" The response is often "ANSI-C" , "Standard C", "Embedded C", "C89" and some times also erroneously "C99" The progression of the C language is shown above. Since 1990 C has been an ISO standard. It is driven by an *international* committee on which the ANSI committee is represented as one among any National Bodies including the UK. C has now been an ISO (internationally) regulated language for 25 years and is defined by ISO/IEC 9899:1990, *Programming languages—C* (C99) and then more recently by ISO/IEC 9899:2011, *Programming languages—C* (C11)

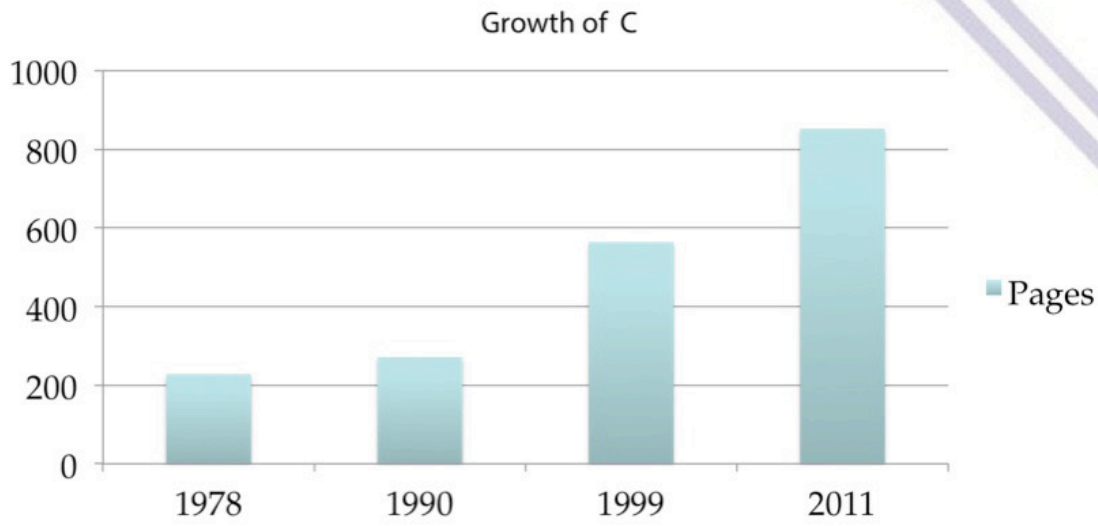
The problem is exacerbated when people ask which books to read to learn C. Often the "Bible" of K&R (Kernighan and Ritchie, The C Programming Language) is cited. Edition 1 is over 36 years old (and dare I say older than most who are asking the question). The second edition, whilst a mere 26 years old, is still older than the version of C the vast majority of C programmers will use.

The other problem is most books on C are a LOT shorter than the standard(s) they are referencing. None of them contain more than a fraction of the ever expanding C standard. The current C standard has more than three times the pages of K&R 2nd Ed and the first ISO-C standard. Two that do contain the standard are on page 13.

Why is this a problem? Simply because the vast majority of C programmers have never seen a copy of the ISO C standard, let alone read it. If you have not read (I am not going to get into "understand") the ISO C standard how can you program in C?

In answer to the question "Which C?" most cross compilers for the embedded market are C95 (with extensions). By 2010 most had evolved from a C90 engine with additions to a C99 engine with omissions. Very few (if any?) compilers for the embedded and critical systems market have been a full implementation of C99. This includes GCC which only loosely follows ISO-C anyway. Then of course the C used for cross compilers for embedded systems will have extensions and restrictions for the target architecture and hardware. This is particularly the case with the standard library; where for a "self hosted" system very little is mandated by the C standards. No one really knows C. This is the Curse of C

MISRA-C:2012 Curse or Cure?



This graph shows the expansion of the C language over more than three decades. Remember that most (but not all) compilers in the embedded market are somewhere between C95 (that is C90 + Amendment 1 and 3 Technical Corrigendum's) and C99. There are signs, in late 2014, that some parts of C11 may find their way into some compilers. A Program Manager for a major embedded compiler company said in 2013 that their compilers would be ISO–C 2011 compliant "where C11 touched C99" and it was something they had already implemented as C99. So while there will be claims of C11 compliance/compatibility I suspect it will more marketing than engineering.

The point is that in practice no working C programmer really understands C and the vast majority have never actually read the C standard: particularly the most important part - the infamous Annex J (in C99 parlance). This addresses portability issues and lists unspecified, undefined and implementation defined behaviour. It covers 25 pages!

This is why MISRA-C is needed: to tame the dangerous parts of the C language and remove many of the common problems. This in turn will cut down debugging time and save money. This was, along with safety, one of the original reasons for MISRA-C. The 90's were the era of "if you don't have the time to do it properly when will you have the time to fix the errors?" The UK car industry had enough problems with time to market without the increasing volume of software in cars adding to it. However the problems are the same in most other industries where software is embedded into products.

MISRA-C:2012 the cure

"To encourage people to pay more attention to the official language rules,

to detect legal but suspicious constructs,

and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his Pcc compiler to produce lint."
Dennis Ritche.
ACM journal 1993

B 9/9/41 D 9/10/11

PhaedruS SystemS

Safely From Conception to Completion
www.phaedsys.com

6 of 143

MISRA-C should be used with static analysis partly because you should not be programming in C without static analysis anyway!

When the first static analyser for C (lint) was built it was to detect legal but suspicious constructs, A LOT of LEGAL C is DANGEROUS according to Denis Ritchie, writing in 1993 about the first lint that was constructed in 1976. Even before K&R wrote the first language reference for C in 1978 and over a decade before ISO C there were problems with C being mis-used.

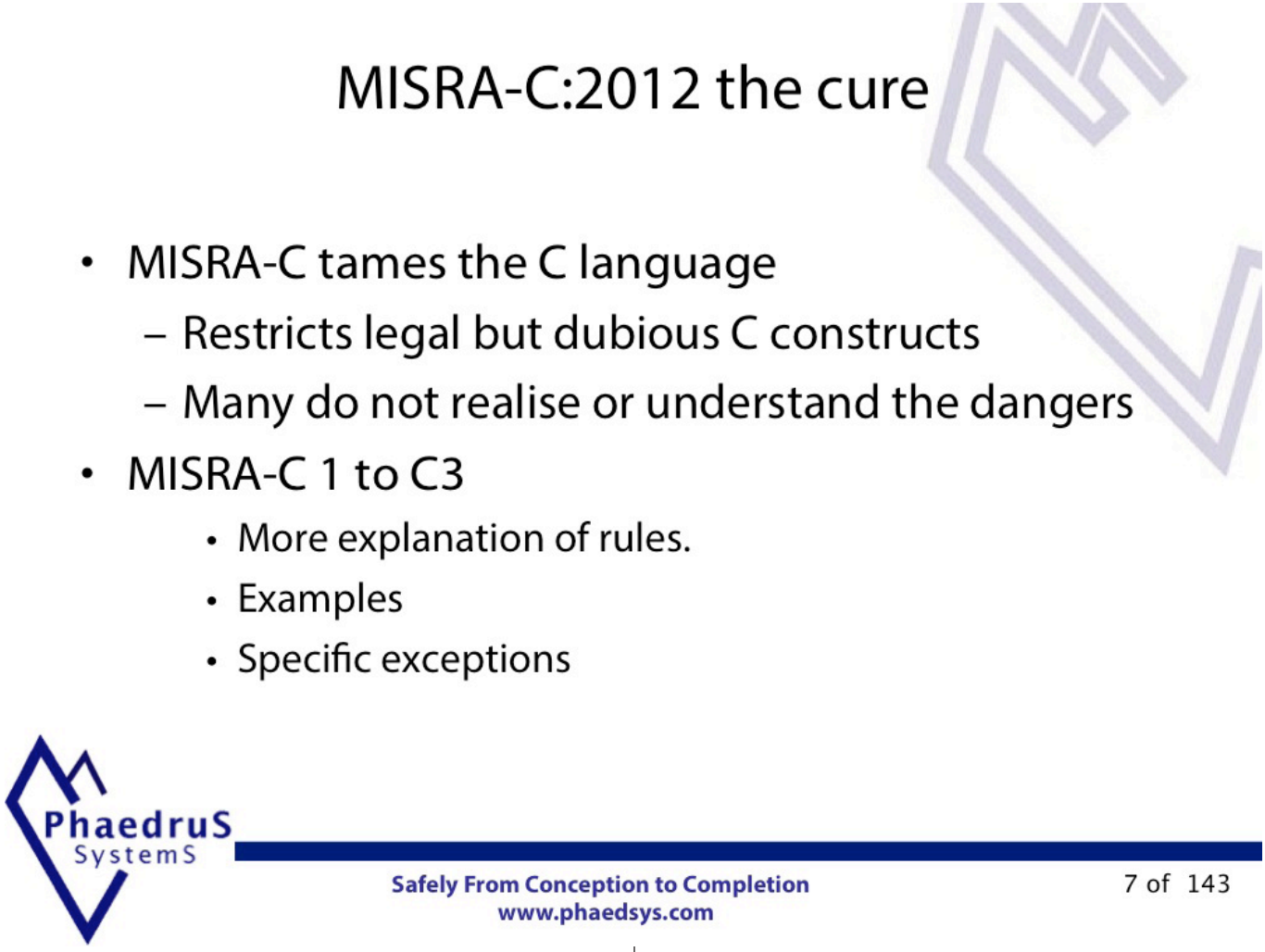
Programmers like to try and prove how clever they are with C. Brian Kernighan had a comment that debugging is twice as hard as writing code. So if you write code to the best of your ability…

It seems that lint (static code analysis) was intended to be part of the standard C compiler chain and certainly it was on UNIX. The problem was it never survived on the leap to the PC development platforms. Many of us did use lint in the 80's on PC's but most never started the habit and it seems universities never pushed it. The culture of "If it compiles it must be OK." started to prevail.

You can read about Steve Johnon, the father of static analysis at: http://en.wikipedia.org/wiki/Stephen_C._Johnson

Since the original lint, high end static code analysers have developed into very powerful code analysers that can produce many code metrics and enforce local coding standards as well as rigorously analyse code with con-figurations for many dialects of C. In the embedded world most compilers have extensions for the hardware architecture, such as specific IO and registers. Before using a Static Code Analyser, check its pedigree to ensure that it can handle the specific dialect of C that you are using.

There are some free static code analysers - but take care as some have not been maintained and so have not kept up with the language or compilers. Also many have not really been properly tested.

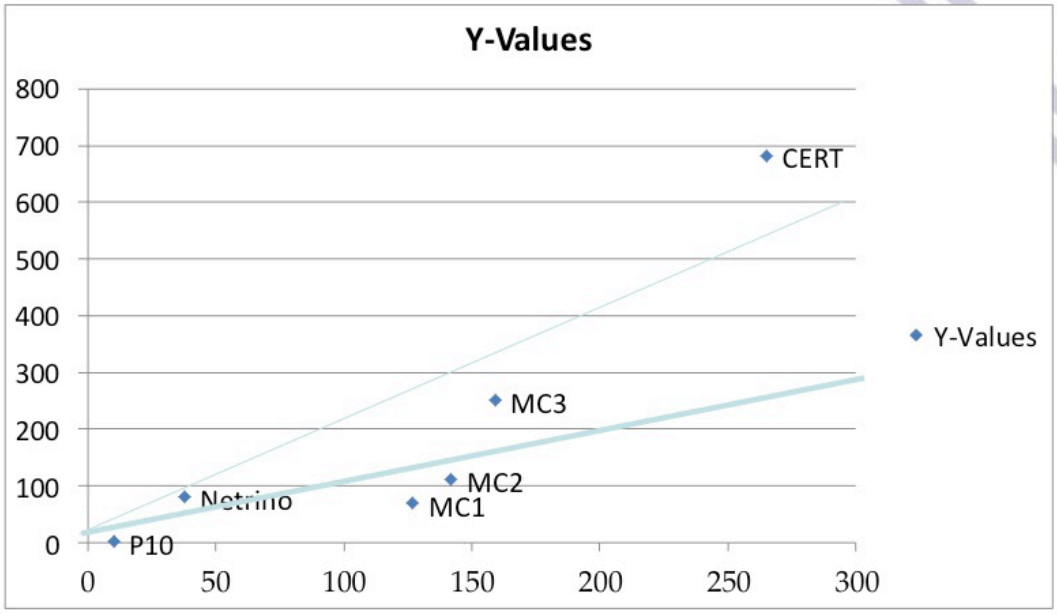


To make C safer, MISRA-C restricts the use of parts of the language, these include legal C that is known to be problematic, misused constructs and the misunderstood parts of C. (This is a larger set than you might think.) These are all areas where many programmers, often erroneously, think they know how the dangerous things work.

As the C language is both evolving and expanding, what may have been correct in C90 is not the same in C99. Even worse the compiler may be a fuzzy mix of C90 and C99, despite claims to be one or the other. This is where a good static analyser, properly configured, pays dividends.

MISRA-C is also evolving and expanding. The evolution is using feedback the tens of thousands of users who are implementing projects using the rules. Static analysis companies have wanted clarification on wording they regarded as ambiguous. So, apart from removing ambiguities MISRA-C had increasing amount of explanation, rationales and examples. It also means that it makes no sense to read only the headline rules.

Rules to size

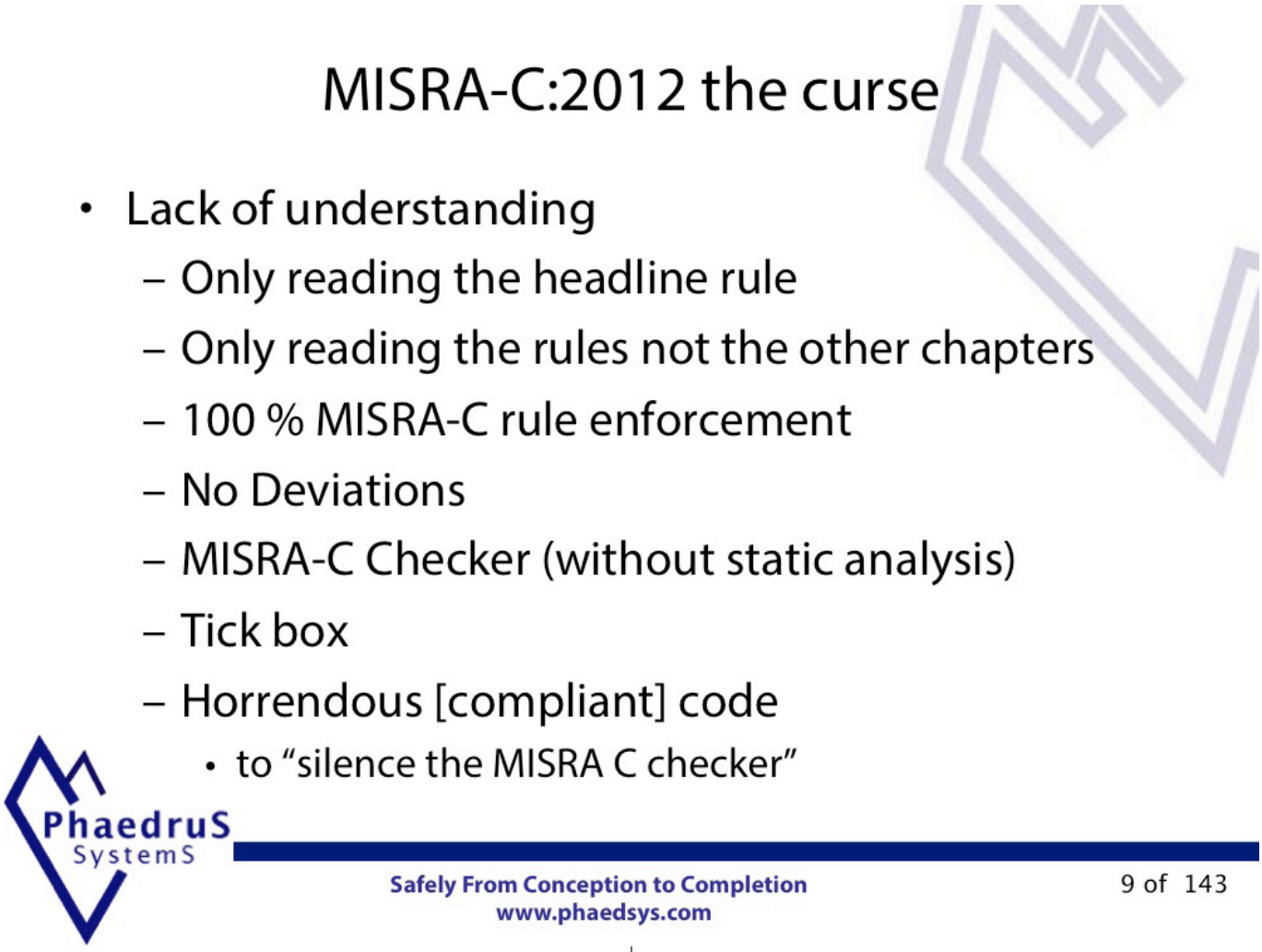


The chart above shows the relationship of the number of pages to the number of rules in some popular coding standards

Power of 10:	10 rules	2 pages,
Neutrino:	38 rules	89 pages
Cert-C:	265 rules	682 pages
MISRA-C1:	127 rules	68 pages
MISRA-C2:	142 Rules	111 pages
MISRA-C3:	159 Rules	256 pages

Since, as we have argued before, most programmers don't have, let alone have read, the ISO C standard, MISRA has to include a lot of explanation of the way C works, or doesn't, in order to explain why the rule is there. However MISRA-C is not a replacement for books on C.

The feedback has also resulted in an increase in guidance on how MISRA-C should be used, particularly for deviations, that is reasoned arguments for not using a particular MISRA-C rule.



The MISRA-C Curse is that many only read the headline rules and not the supporting text for the rule. MISRA-C:2012 has attempted to change this by shortening the headline rule and adding a rational and amplification, making rule usable only by reading all three parts (and, of course, any exceptions). A further problem is that many only read the chapter containing the rules and not the rest of the document, which explains the how and why of implementing the rules, without which MISRA-C compliance can not be claimed. This is required reading.

One of the results of not reading and understanding the context in the supporting chapters is project managers calling for 100% MISRA-C rule enforcement with no deviations. This is not possible. If MISRA-C is used there *WILL* be deviations: it is not possible to do otherwise. The follow on is that some look for a "MISRA-C checker" without doing static code analysis. Whilst some of the MISRA-C rules are not statically enforceable the majority are, and are an enhancement to static code analysis. This is why most static code analysis tools can enforce a large number of the MISRA-C rules (typically 80% of them). It is pointless to either manually check for MISRA-C compliance or use a tool that does not do static analysis at the same time.

A "tick box" culture to implementing MISRA-C has developed. As well as giving the programming team many problems, it can also produce horrendous source code that while technically meeting the rules negates their spirit or intent. There have been many cases of code written to "silence the MISRA checker" rather than address the underlying problem.

The MISRA-C team are often asked to give an opinion on a constructed problem. When the answer is "don't do it that way in the first place" they are then asked to rule on the specifics of the example. The questioner is looking for a way to circumvent the letter of the MISRA rule and wants to avoid what would be the good engineering practice of redesigning the code to work in a better way.

MISRA-C:2012 the curse

Deviations

Blind adherence to the letter without understanding is pointless. Anyone who stipulates 100% MISRA-C coverage with no deviations does not understand that they are asking for.

In my opinion they should be taken out and… Well…

just taken out.

Chris Hills, Member of MISRA C Working Group
MISRA Matters Column in MTE June 2012

Only 6% of the MISRA-C rules are Mandatory: that is rules that are applicable 100% of the time. Therefore we hope that 99.9999% of MISRA-C users will deviate the rules in an appropriate manner.

We are regularly asked about deviations. Firstly we are asked "How do we deviate?" This is discussed in the two papers referenced at the end of this paper.

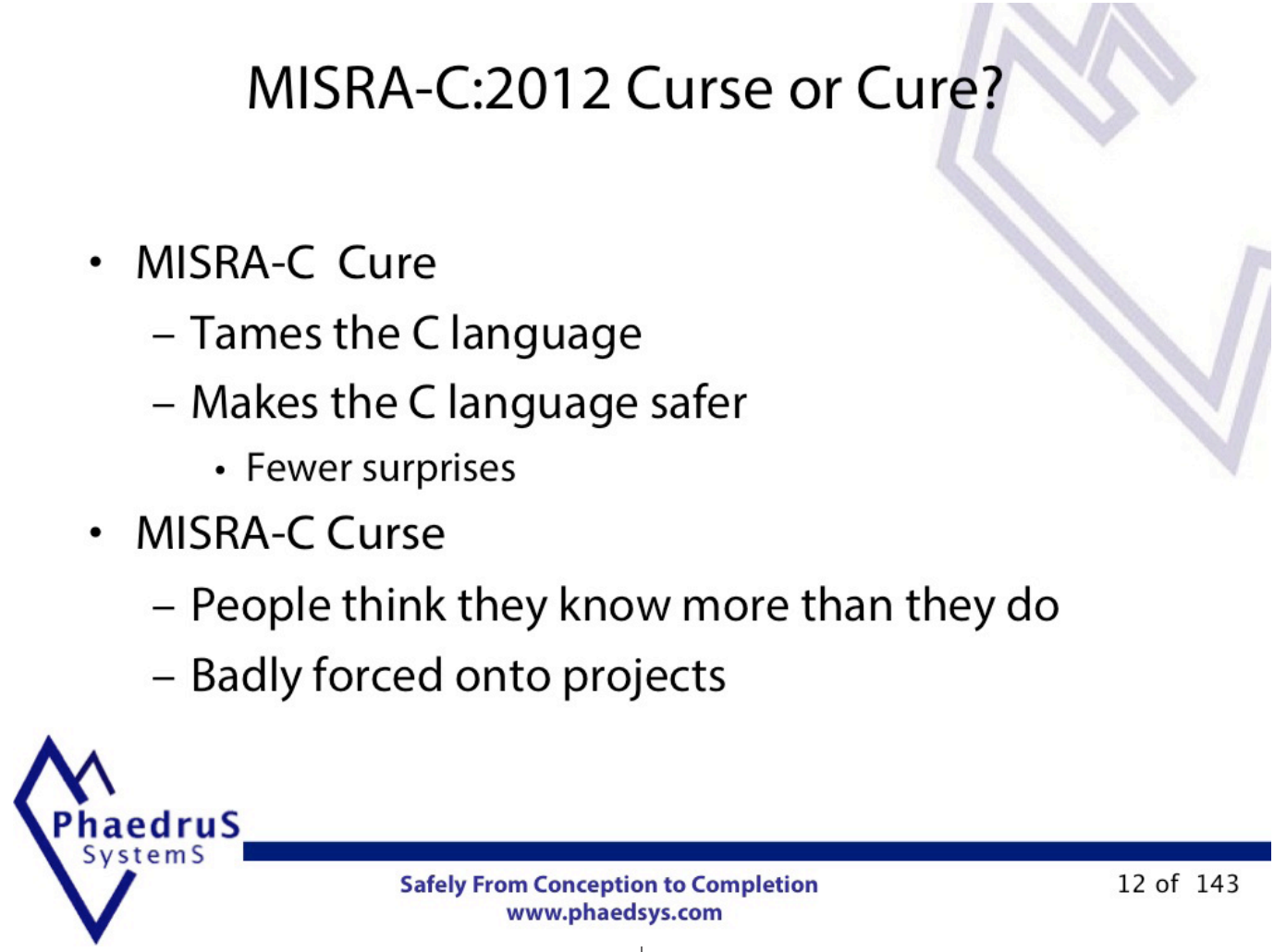
Secondly we hear from people who have been told, "100% MISRA-C with no deviations. [TICK]" This instruction is always from people who don't understand what MISRA-C is or how to implement it.

This is one of the places where MISRA-C can be counter productive. A manager demanding 100% compliance doesn't realise that he is dangerously handicapping the project. The team will have to fight with the standard and resort to time consuming and, in some cases, dangerous, tricks to get round the warnings from the code analysers. They will waste time and produce hideous and less efficient code.

In most presentations on MISRA-C by MISRA-C team members and tool vendors with MISRA-C checkers, the speaker will tell the audience you are going to have to deviate some rules and this is NOT a tick box or simple decision. It requires thought and consideration. YOU have to do this on YOUR responsibility and it will be different for every project.

A4 size Copies of this slide are available signed for your manager's office wall!

Deviations WILL be required. Just as the rain must fall (but too much is a flood).

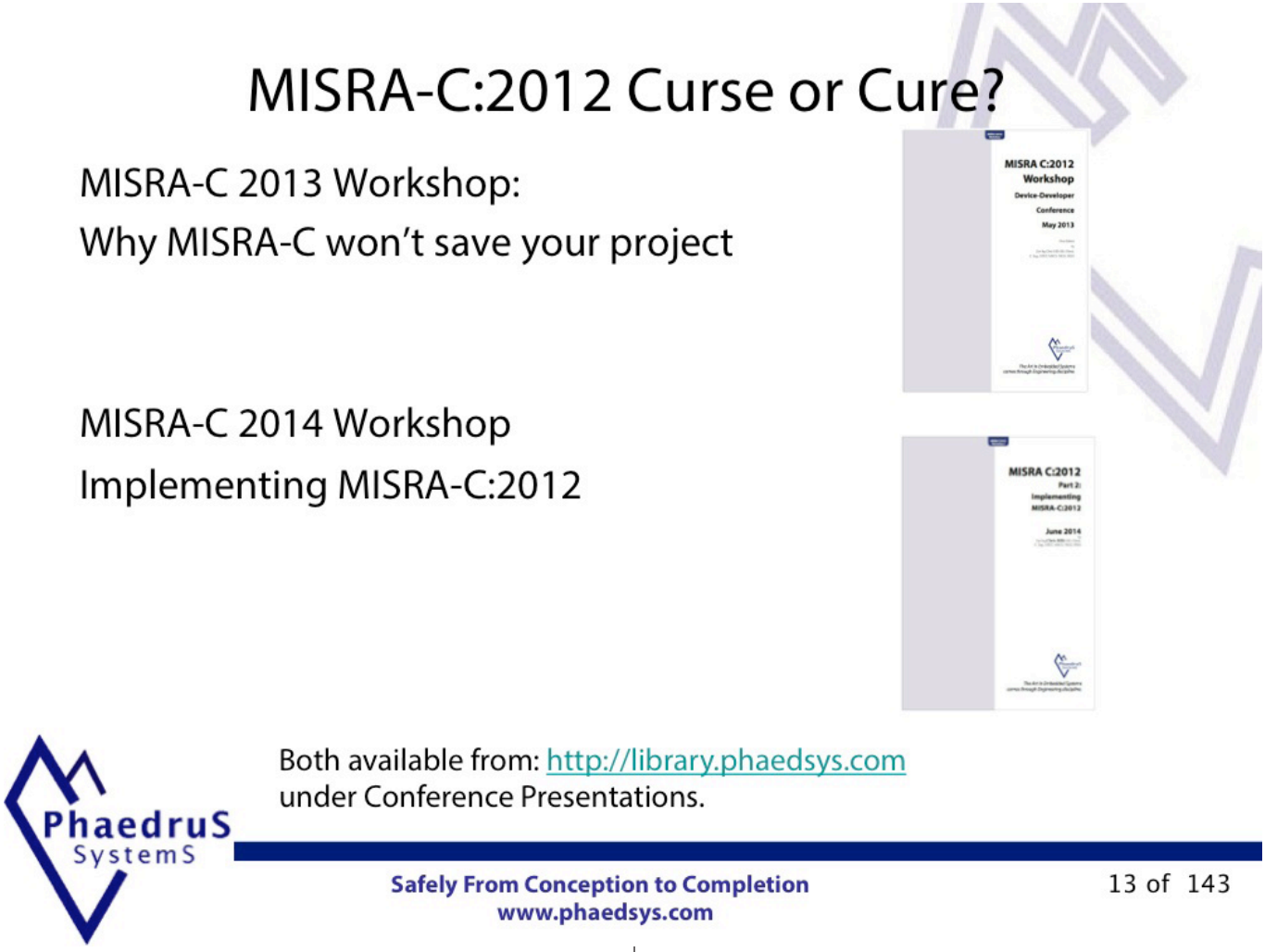


MISRA-C is the cure that tames the C language and helps to make it more predictable with fewer surprises. The support for MISRA-C in static code analyzers encourages the use of static analysis and automated source code checking by rigorous tools. This can cut the test and debug phase drastically. Over the years reports and studies have shown anything between 25% and 40% savings on project time by using a good static code analyzer and MISRA-C. However, using MISRA-C without a static code analyzer to check the source is pointless, and will probably do more harm than good and certainly not give the gains mentioned above.

The MISRA-C curse is that people think they know more than they do about C. I know a highly experienced C trainer who says he learns more about C every time a new MISRA-C appears.

We on the MISRA-C working group are also constantly learning, despite most of us being involved in C standards, and in making C tools. We have, on average over 25 year's industrial experience. The MISRA-C team draws on experience in two ways. Some members are on the standards panels and some members are making some of the world's leading C analysis tools though all have a background in writing code on real projects. We also have direct and regular contact with the developers of the world's leading C compilers. Some time we may not know about a feature of C but we always know an expert who does. (Come to that often when we think we know we still check!) On the other hand we have lot of contact with real implementations in our company's or our customer's projects. We see what actually happens on millions of lines of C over thousands of projects.

The other part of the MISRA Curse is that MISRA-C is often badly implemented on projects. This is partly due to management not understanding that deviations are not a bad thing, not recognizing the need for a compliance matrix etc, nor the need for static analysis tools. It is also partly due to programmers not always understanding the rules and/or C as well as they think they do. This can mean that badly implemented MISRA-C can do as much harm as good…. But all is not lost!



This presentation was a short look, just 12 minutes on the Advanced Engineering 2014 show floor. As a perspective, Feabhas, the training company (www.feabhas.co.uk) present MISRA-C courses. The "over view" takes a whole day and the full course is four full days!!!

The two 45 minute presentations above provide a halfway house. These were given as part of the MISRA-C workshops in the 2013 and 2014 Device Developer Conferences.

The 2013 document explains why MISRA-C might do more harm than good on a project, outlining some of the more common problems in trying to use the guidelines. The 2014 document goes on to talk about how to implement MISRA-C in a way that it makes maximum improvement.

The two presentations above are available from the Phaedrus Systems Library under Conference Presentations

http://www.phaedsys.com/library/presentations.html

Bibliography

BSI, The C Standard (incorporating Technical Corrigendum 1), Wiley 2003, 978-0-470-84573-8 (This is the complete C99 with the rationale)

Derek Jones, The New C Standard: An Economic and Cultural Commentary, 2008 (also C99) free from

http://www.coding-guidelines.com/cbook/cbook1_1.pd

MISRA-C 2013 Workshop: Why MISRa-C won't save your project

http://www.safetycritical.info/library/presentations/MISRA-C3-0001.pdf

MISRA-C 2014 Workshop: Implementing MISRa-C:2012

http://www.safetycritical.info/library/presentations/MISRA-C3-0002.pdf

MISRA C:2012 Curse or cure
Advanced Engineering Conference
November 2014

First edition November 2014

Phaedrus Systems Library

The Phaedrus Systems Library is a collection of useful technical documents on development. This includes project management, integrating tools like PC-lint to IDE's, the use of debuggers, coding tricks and tips. The Library also includes the QuEST series.

Copies of this paper (and subsequent versions) with the associated files, will be available with other members of the Library, at:

http://library.phaedsys.com

PhaedruS SystemS

The Art in Embedded Systems comes through Engineering discipline.