# The MISRA C Coding Standard: A Key Enabler for the Development of Safety- and Security-Critical Embedded Software

Roberto Bagnara, Abramo Bagnara, Patricia M. Hill

bugSeng
http://bugseng.com

& Applied Formal Methods Laboratory
Department of Mathematical, Physical and Computer Sciences
University of Parma, Italy

16th Automotive Software Workshop, CRF, Orbassano (TO), Italy, February 21st, 2019

# Outline I

# Disclaimer

I am a member of the *MISRA C Working Group* and of ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*

however

the views expressed in this presentation and the accompanying paper are mine and my coauthors' and should not be taken to represent the views of either working group

# Advantages of C

- C compilers exist for almost any processor
- C compiled code is very efficient and without hidden costs
- C allows writing compact code (many built-in operators, limited verbosity, ...)
- C is defined by an ISO standard
- C, possibly with extensions, allows easy access to the hardware
- C has a long history of usage in critical systems
- C is widely supported by tools

# Disadvantages of C

ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*, has always been faithful to the original spirit of the language

I Trust the programmer
II Let the programmer do anything
III Keep it fast, even if not portable
IV
V

**Bad for safety and security!**

# What is "Behavior"

## ISO/IEC 9899:1999 TC3 (N2156) 3.4

**behavior**
external appearance or action

## As-if rule

The compiler is allowed to do any transformation that ensures that the "observable behavior" of the program is the one described by the standard

True in C, but also in C++, Rust, Go, OCaml, ...

# What is "Undefined Behavior"

ISO/IEC 9899:1999 TC3 (N2156) 3.4.3

**undefined behavior**
behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

No requirements means absolutely no requirements: crashing, erratic behavior of any kind, formatting the hard disk!

Normally it means the compiler assumes undefined behavior does not happen

If it does happen, the programmer has violated the contract: warranty void!

# Undefined Behavior: Examples

- The program attempts to modify a string literal (6.4.5)
- The string set up by the `getenv` or `strerror` function is modified by the program (7.20.4.5, 7.21.6.2)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *path = getenv("PATH");
    path[0] = '\0'; /* Disable path search. */

    /* How does the program behave? */
}
```

# Undefined Behavior: Examples (cont'd)

- The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8)
- A trap representation is read by an lvalue expression that does not have character type (6.2.6.1)

```
int main() {
  int a;
  if (a > 0) /* Undefined behavior.  Bit-trap?  Maybe. */
    return 1;
  else
    return 0;
}
```

# What is "Unspecified Behavior"

ISO/IEC 9899:1999 TC3 (N2156) 3.4.4

**unspecified behavior**
use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

# Unspecified Behavior: Example

- The order in which the operands of an assignment operator are evaluated (6.5.16)

```
int *f(int **pp) {
  ++(*pp);
  return *pp;
}

int main() {
  int a[] = {  0,  1,  2,  3 };
  int b[] = { 10, 11, 12, 13 };
  int *p = a;
  int *q = b;

  *( f(&p) ) = q[ *p ];
}
/* Contents of a: 0, 10, 2, 3 or
                  0, 11, 2, 3 ? */
```

# What is "Implementation-Defined Behavior"

ISO/IEC 9899:1999 TC3 (N2156) 3.4.1

**implementation-defined behavior**
unspecified behavior where each implementation documents how the choice is made

# Implementation-Defined Behavior: Example

- Which of signed char or unsigned char has the same range, representation, and behavior as "plain" char (6.2.5, 6.3.1.1)

```
#include <limits.h>
#include <stdio.h>

int main() {
  if (CHAR_MIN == 0)
    fputs("unsigned char\n", stdout);
  else
    fputs("signed char\n", stdout);
  /* What is the output? */

  /* Compliant C implementations document this behavior. */
}
```

# Why?

We described:

- Undefined behavior
- Unspecified behavior
- Implementation-defined behavior
- (and we glossed over locale-specific behavior)

Why is the standardized language not fully defined?

- Because implementing compilers is easier
- Because compilers can generate faster code

# UB: Signed Integer Overflow

The behavior is undefined when:

> *An exceptional condition occurs during the evaluation of an expression (6.5)*

```
int always_true(int v) {
  return (v + 1 > v) ? 1 : -1;
}
```

Can be compiled as

```
int always_true(int v) {
  return 1;
}
```

because the compiler assumes `v + 1` does not overflow

# UB: Modifying String Literals

The behavior is undefined when:

> *The program attempts to modify a string literal (6.4.5)*

Example: in a program there are literals `"Tail"` and `"HeadTail"`

The compiled program can store in memory only `"HeadTail"` and return the pointer to the fifth character as `"Tail"`

Changing one string may also change the other, but the compiler can assume this will never happen

# UB: Shifting Too Much

The behavior is undefined when:

> *An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7)*

```
uint32_t i = 1;
i = i << 32;  /* Undefined behavior. */
```

Strange: if I push 32 or more zeros from the right the result should be zero, right?

# UB: Shifting Too Much Example

From Intel64 and IA-32 Architectures Manual, page 1706 section "IA-32 Architecture Compatibility":

> *The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.*

Basically, this means that in those machines

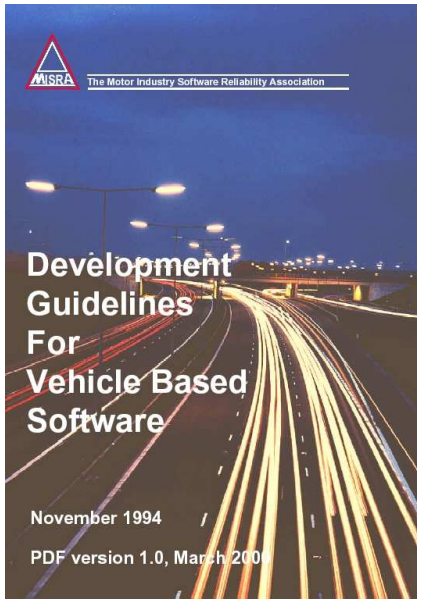
```
i = i << 32;            /* This is equivalent to... */
i = i << (32 & 0x1F);   /* ... this, i.e., ...      */
i = i << 0;             /* this, which is a no-op.  */
```

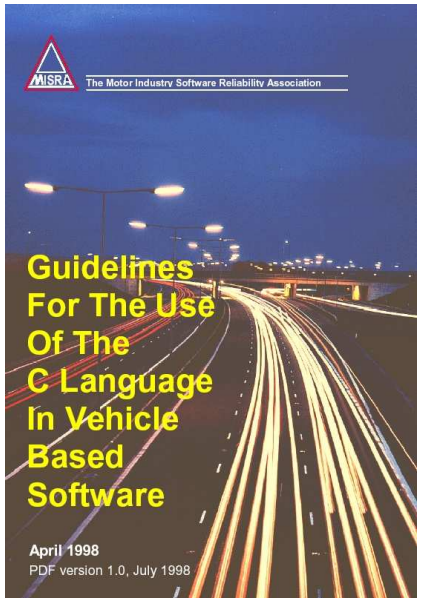
C leaves the behavior undefined for speed and ease of implementation

# The MISRA Project

- Started in 1990
- Mission: To provide world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software
- The original project was part of the UK Government's "SafeIT" programme
- Now self-supported
- HORIBA MIRA provides the project management support

# Original MISRA Publications



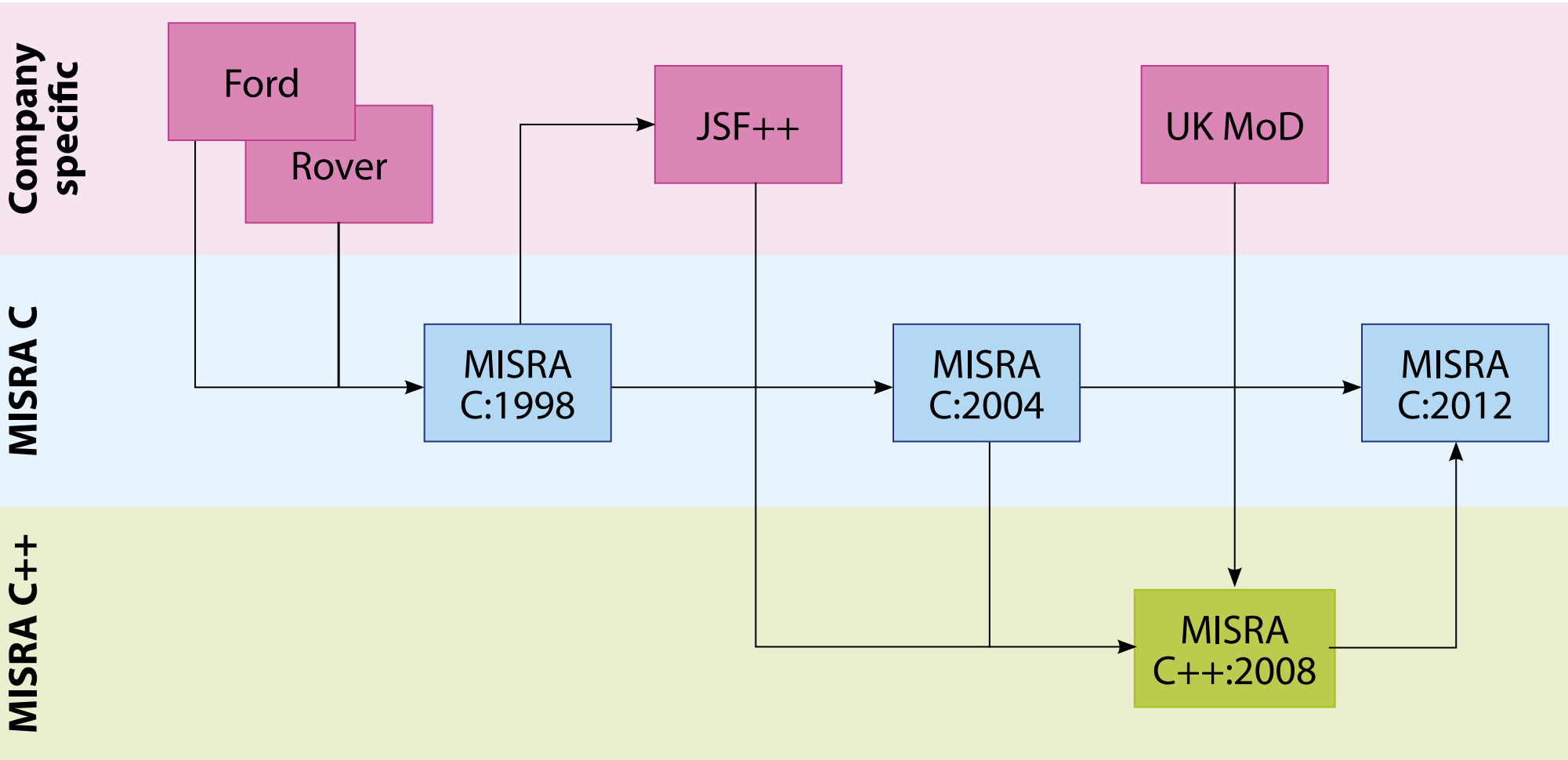




November 1994: *Development guidelines for vehicle based software* (The MISRA Guidelines)

- The first automotive publication concerning functional safety, more than 10 years before work started on ISO 26262!

April 1998: *Guidelines for the use of the C language in vehicle based software* (MISRA C)

# History of MISRA C/C++ Guidelines



# Strength and Weakness of C

The weakness of the C language comes from its strength:

- Ease of writing efficient compilers for almost any architecture $\Longrightarrow$ non-definite behavior
- Efficient code with no hidden costs $\Longrightarrow$ no run-time error checking
- Many compilers, defined by an ISO standard (must standardize existing practice, many vendors, backward compatibility) $\Longrightarrow$ non-definite behavior
- Easy access to the hardware $\Longrightarrow$ easy to shoot your own foot
- Compact code $\Longrightarrow$ the language can be easily misunderstood and misused

# Language Subsetting

Several features of C do conflict with both safety and security

For safety-related applications, language subsetting is crucial

Mandated or recommended by all safety- and security-related industrial standards:

- IEC 61508
- ISO 26262
- CENELEC EN 50128
- RTCA DO-178C

The most authoritative language subset for the C programming language is MISRA C

# Presentation of the Guidelines

| | |
|---|---|
| Rule 5.6 | A *typedef* name shall be a unique identifier |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

## Amplification

A *typedef* name shall be unic
the same *typedef* name are d
that *header file* is included in

## Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

On the left, a unique identifier for the guideline composed by a classification as “Rule” or “Directive” followed by a dot-decimal sub-identifier; the remaining text is called the headline of the guideline

# Presentation of the Guidelines (cont'd)

Rule 5.6 A *typedef* name shall be a unique identifier

Category: Required

Analysis: Decidable, System

Applies to: C90, C99

One of "Mandatory", "Required" or "Advisory"

A pair of the form *Decidability*, *Scope*: the former is one of "Decidable" or "Undecidable", the latter is one of "System" or "Single Translation Unit"

One or more of "C90" and "C99" separated by comma

## Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that h

## Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

# Scope of Analysis

Rules are classified according to the amount of code that needs to be analyzed in order to detect all violations

**Single Translation Unit:** All violations within a project can be detected by checking each translation unit independently

**System:** Identifying violations of a rule within a translation unit requires checking more than the translation unit in question, if not all the source code that constitutes the system

# Presentation of the Guidelines (cont'd)

Rule 5.6 A *typedef* name shall be a unique identifier

Category Required

Analysis De

Applies to C9

A more precise description of the guideline: this is normative!

## Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that *header file* is included in multiple source files.

## Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

# Presentation of the Guidelines (cont'd)

| Rule 5.6 | A *typedef* name shall be a unique identifier |
|---|---|
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

## Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* na nade in a *header file* and that *header file* is in

The reason why the guideline exists

## Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

# Presentation of the Guidelines (cont'd)

Rule 5.6 A *typedef* name shall be a unique identifier

Category Required

Analysis Decidable, System

Applies to C90, C99

## Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* na ader file and that *header file* is inc

A description of the situations in which the rule does not apply

## Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

# Presentation of the Guidelines (cont'd)

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Examples showing compliant and non-compliant code

## Example

```
void func ( void )
{
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t;     /* Non-compliant - reuse */
  }
}

typedef float mass;

void func1 ( void )
{
```

## See also

Rule 5.7

# Presentation of the Guidelines (cont'd)

## Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

## Example

```
void func ( void )
{
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t;     /* Non-compliant - reuse */
  }
}

typedef float mass;

void func1 ( void )
{
```

Reference to related guidelines

## See also

Rule 5.7

# Presentation of the Guidelines (cont'd)

| Dir 1.1 | Any implementation-defined behaviour on which the output of the program depends shall be documented and understood |
|---|---|

C90 [Annex G.3], C99 [Annex J.3]

Reference to one or more published sources to be consulted for a fuller understanding of the rationale

**Category** Required

**Applies to** C90, C99

## Amplification

Appendix G of this document lists, for both C90 and C99, those implementation-defined behaviours that:

- Are considered to have the potential to cause unexpected program operation, and
- May be present in a program even if it complies with all the other MISRA C guidelines.

All of these implementation-defined behaviours on which the program output depends must be:

- Documented, and
- Understood by developers.

*Note:* a conforming implementation is required to document its treatment of all implementation-defined behaviour. The developer of an implementation should be consulted if any documentation is missing.

# Software Development Process

The highest payoff from the adoption of MISRA C is achieved when it is used in the framework of a documented software development process

The process must ensure, e.g.:

- that software requirements are complete, unambiguous and correct
- that design specifications reaching the coding phase are correct, consistent with the requirements and do not contain any other functionality
- that object modules produced by the compiler behave as specified in the designs
- that object modules have been tested, individually and together, to identify and eliminate errors

# Software Development Process (cont'd)

MISRA C should be used before code reaches the review and unit testing phases...

...otherwise a lot of rework and retesting has to be expected

Full requirements for safety-related software development processes are outside the scope of MISRA C

Examples of development processes may be found in, e.g.:

- IEC 61508
- ISO 26262
- RTCA DO-178C
- CENELEC EN 50128
- IEC 62304

# MISRA C: Error Prevention, Not Bug Finding

MISRA C cannot be separated from the process of documented software development it is part of

The use of MISRA C in its proper context is part of an error prevention strategy which has little in common with bug finding

Violation of a guideline is not necessarily a software error

- E.g., there is nothing intrinsically wrong about converting an integer constant to a pointer when it is necessary to address memory mapped registers or other hardware features
- However, such conversions are implementation-defined and have undefined behaviors, so Rule 11.4 suggests avoiding them everywhere apart from the very specific instances where they are both required and safe

# MISRA C: Error Prevention, Not Bug Finding (cont'd)

The deviation process is an essential part of MISRA C

The point of a guideline is not "You should not do that"

The point is: "This is dangerous: you may only do that if

1. it is needed
2. it is safe
3. a peer can easily and quickly be convinced of both 1 and 2"

One useful way to think about MISRA C and the processes around it is to consider them as an effective way of conducting a guided peer review to rule out most C language traps and pitfalls

# MISRA C: Error Prevention, Not Bug Finding (cont'd)

The attitude with respect to incompleteness is entirely different between the typical audience of bug finders and the typical audience of MISRA C

- Bug finders are usually tolerant about false negatives and intolerant about false positives
- This is not the right mindset for checking compliance with respect to MISRA C: false positives are a nuisance, but false negatives imply other methods will have to be used to ensure compliance

# MISRA C: Error Prevention, Not Bug Finding (cont'd)

Another aspect that places MISRA C in a different camp from bug finding has to do with the importance MISRA C assigns to reviews:

- code reviews
- reviews of the code against design documents
- reviews of the design documents against requirements

(This fact has some counterintuitive consequences on the use of static analysis)

Many MISRA rules have to do with code readability and understandability

# Adoption

The highest payoff from the adoption of MISRA C is achieved when it is adopted at the very beginning of a project...

...and it is systematically enforced with the help of a high-quality tool

Imposing MISRA C on an existing code base with a proven track record may be counterproductive if not done properly...

...this requires significant expertise and tools of even higher quality (powerful deviation mechanisms, baselining, ...)

# Manual vs Semiautomated Verification

It is entirely possible to manually verify code for compliance to MISRA C

The cost of doing that properly is of course enormous

Tools are highly recommended to semiautomate the check for compliance

Manual activities remain, such as:

- Initial tool configuration
- Tool configuration for deviation

# Manual vs Semiautomated Verification (cont'd)

A good tool will do a thorough job of automatically verifying compliance for most MISRA C guidelines...

...but not all of them:

- undecidable rules
- directives
- limitations of the tool (complexity-precision trade-off, incomplete handling of extensions, ...)
- issues with the project being analyzed (unavailability of part of the source code, extensive use of language extensions, ...)

The remaining manual activities, and peer review, are greatly facilitated by the level of partial compliance that can be quickly achieved by using a good tool

# Tool Configuration: C is a Large Family of Languages

In C99, there are 112 implementation-defined behaviors

As each i.d.b. can be defined in 2 or more ways, there are more than $2^{112} \approx 5 \times 10^{33}$ possible languages

Actually, choosing integer and floating-types in $\{8, 16, 32, 64\}$ brings us to more than $10^{36}$ possible languages (dialects of C)

Alexander's star: $7.24 \times 10^{34}$ different positions

# C is a Large Family of Languages (cont'd)

Generally speaking, a given compiler can implement, via options, several such dialects of C

For an extreme case, GCC/x86_64 implements, via options, hundreds of dialects of C

As a consequences, the tool must adapt to the particular dialect implemented by that compiler with that set of options (possibly for each translation unit)

Further consequence: changing even one compilation option may have important consequences

We have seen many projects whose MISRA C compliance was completely led astray due to misconfiguration of the actual language dialect implemented by the toolchain

# Predefined Macros

Another aspect that might require careful configuration of the tool has to do with predefined macros

Some of them may be influenced by compiler options, which may be given

- on the command line
- in environment variables
- in files

Failing to capture the predefined macros correctly may result into the wrong code being analyzed

This is the cause of many headaches (not to count where/how header files are searched, compiler intrinsics and other extensions, the linker, the librarian, ...)

# Tool Configuration Related to the Toolchain

There are three possibilities:

1. the tool is integrated in the compiler/linker; all the required information is there but, generally speaking:
   - such tools are not very good (limited audience, limited investments, limited testing)
   - you change compiler and you will have to start from scratch
2. the tool assumes you carefully configure it:
   - very error-prone
   - someone changes the compiler options and you may have to redo the configuration
3. the tool is smart enough

# True Compliance Requires Staff Competence

Staff competence is a crucial requirement in order to carry out the activities that allow describing a project as "MISRA Compliant"

Without a proper understanding of C pitfalls and of the reasons behind each of the MISRA guidelines, developers often:

- perceive the adoption of the guidelines as a useless burden
- misunderstand messages output by the tool and do not know what should be done
- are unable to recognize false positives
- change the code by trial-and-error in an attempt to silence the tools (code quality may decrease!)

# Conclusion

C is the most used language for the programming of embedded systems

The advantages of C come with corresponding disadvantages that severely impact safety and security: language subsetting is crucial!

MISRA C is the most authoritative subset of C for the development of high-integrity embedded systems

MISRA C is integral part of a software development process, and its adoption is radically different from bug finding

Good tools and proper formal training of personnel enables a smooth and successful adoption of MISRA C into an organization

# The End


Questions?
roberto.bagnara@bugseng.com
bagnara@cs.unipr.it
bugseng