How I learned to stop worrying and love MISRA

Loïc Joly

MUREX

CODE RECKONS

Science to the CORE

sonarsource

How I learned to stop worrying and love MISRA

Loïc Joly - loic.actarus.joly@gmail.com

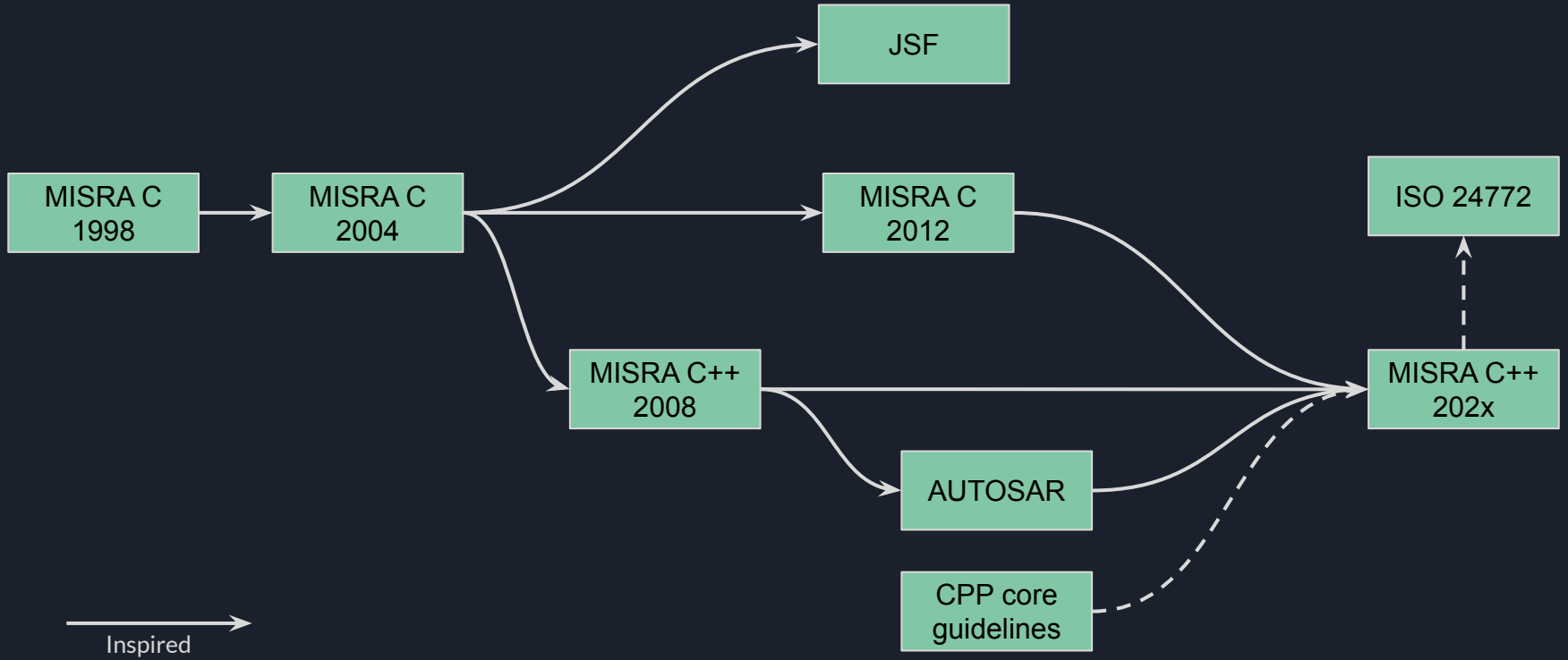

I used MISRA…
I cannot crash

What is MISRA

- A set of guidelines for safety-critical software
- Has its roots in the automotive industry
- Among many existing sets of guidelines, one of the most followed
- Current version for C++: MISRA C++ 2008

No, it's not a typo...

Simplified history of MISRA



What does a MISRA rule look like

- A title
- A category: Mandatory, Required, Advisory
- Analysis: Decidable or not, Single translation unit or System
- Amplification
- Rationale
- Examples

Example of a MISRA C++2008 rule

Rule 10.3.2 Each overriding virtual function shall be declared with the *virtual* keyword.

Category Required

Rationale

Declaring overriding virtual functions with the *virtual* keyword removes the need to check the base class to determine whether a function is virtual.

Example

```
class A
{
public:
   virtual void g();
   virtual void b();
};

class B1 : public A
{
public:
   virtual void g();  // Compliant     - explicitly declared "virtual"
   void b();          // Non-compliant - implicitly virtual
};
```

Why did I hate MISRA with a passion?

Some context

- I love C++
  - I love teaching it
  - I love spreading good practices
- I’m writing static analysis tools
  - Implementing specific rules
  - Specifying specific rule
  - Our tools can check some MISRA rules

Why did I hate MISRA with a passion?

- Very often, when implementing a MISRA rule, we were unhappy
  - The rule felt odd
  - The rule make people write C++ in a C style
  - The rule make people write overly verbose code
  - Code following the rule can often be inferior to code ignoring it
  - Everything is outdated. Good C++ today is different from good C++ in 2008
  - Even back in 2008, some rules were highly questionable
  - Some rules go against the spirit of C++
- Implementing those rules led to a high cognitive dissonance to me

Some examples

6.6.5 A function shall have a single point of exit at the end of the function.

We all know that single-entry/single-exit leads to code more difficult to read, with more internal state stored in variables.

Additionally, this seems inconsistent with the fact that MISRA allows exceptions.

Some examples

14.7.2 For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.

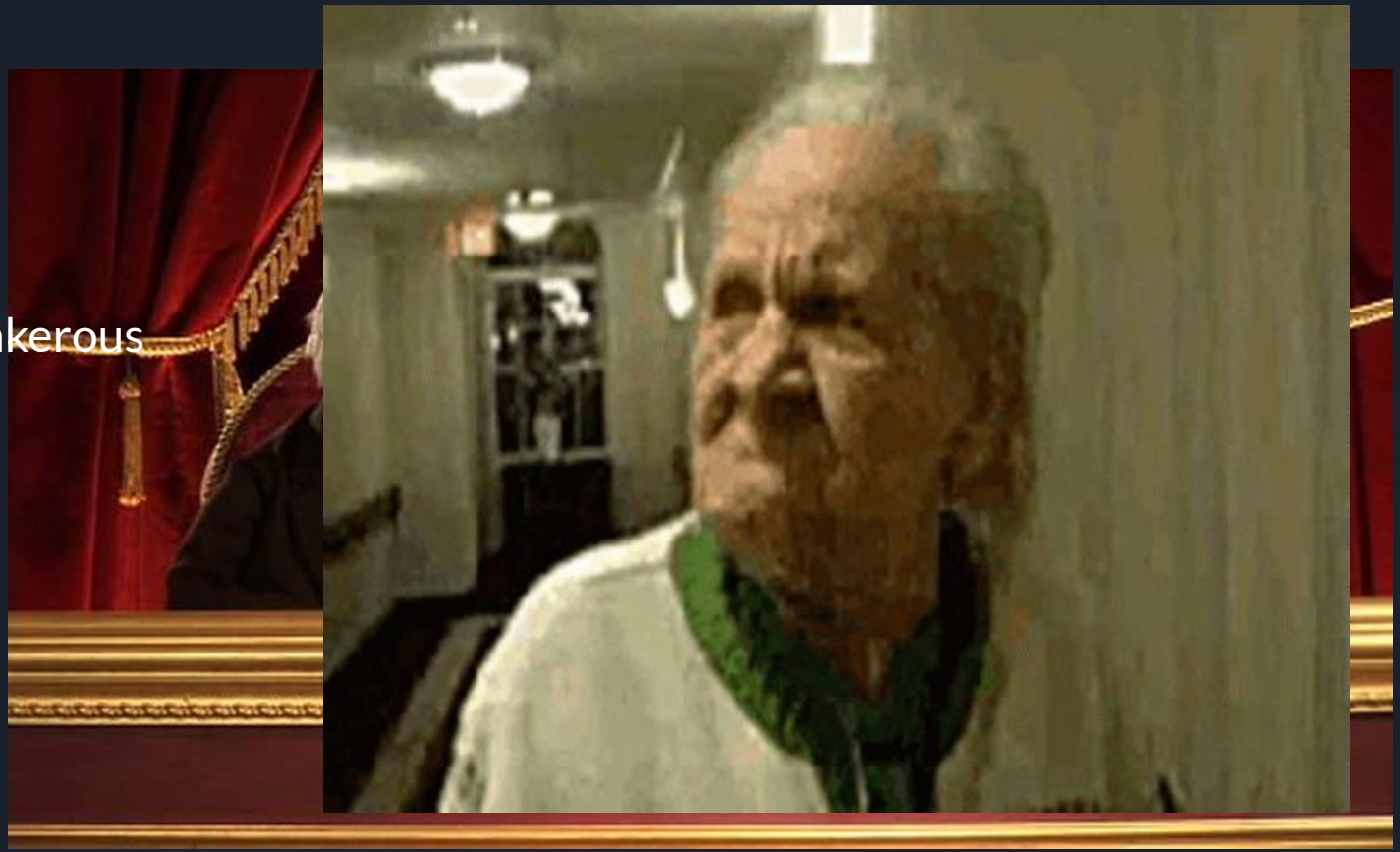
This goes against the philosophy of C++ that a class template can have different features, depending on properties of the type it is instantiated with. For instance, this rule would not allow to instantiate :

```
std::list<std::complex<double>>
```

What did I do?

I became grumpy

And even cantankerous



But then...

- I learned that the MISRA C++ working group was meeting again
- With the goal of publishing an updated version
- They were welcoming people to join them
- So I joined...
- ...and I learned

Design goals for MISRA 1/3

- Focus on safety only
  - If you don’t work on safety-critical software, it may be too much of a pain to follow
  - It does not care about other aspects, and should be mixed with other rules that target for instance performance, style, domain specific issues, security…
- Guidelines should be tool-checkable
  - Different tools should have the same opinion about some code
  - Rule amplification is more for tool vendors than for users
- Guidelines should prevent bad code, not promote good code
  - For instance, no guideline about using strong types (but you should anyways)
  - But rules against things that happen when using fundamental types

Design goals for MISRA 2/3

- Undefined behaviour should be prevented as much as possible
  - Even if it restricts what kind of code can be written
  - Even if it is slightly painful
- Explicit is better than terse
- Decidability is important
  - Sometimes, a set of rule works together
  - One non-decidable to catch the real issue
  - Several decidable that make sure that code is written in a way the the real issue won't happen (those might be more constraining)
  - For example:
    - You should not access object out of its lifetime (non-decidable)
    - You should not return a reference to a local variable (decidable)
    - You should not use unions (decidable, much broader, but deemed acceptable)

Design goals for MISRA 3/3

- When MISRA says : You should not write code that way
  - The goal is not to prevent you from writing code that way
  - The goal is that if you do, it should be validated
  - Exception: Mandatory rules
- Deviation process
  - You should explain why it is safe to deviate from the rule in that case
  - In many environments, it also means that someone else should agree with you

Case study: for loops in MISRA 2008

- It is difficult to write a loop and make sure it will end
- A group of rules for for-loops that enforce constant increment of loop variable and fixed final value
- For instance: The *loop-counter* shall not be modified within *condition* or *statement*.
- Difficult notion of *loop-counter*
- Those rules prevent writing useful code!
- Consequence: Everybody uses the less constrained while loops
- Even when a for loop would have been clearer

Case study: for loops in MISRA 202x

- It is difficult to write a loop… (nothing changed)
- One rule: Legacy iteration statements should not be used
- Also covers while loops and do-while loops
- Common advice: Prefer algorithms and range-based for loop
- If you are writing your own loop, you are not in the easy case
- It does not mean you cannot use legacy iteration statements
- It means each time you use one, you'll have to:
  - Justify why other solutions were inferior
  - Validate that your loop is correctly written
- Exceptions for common patterns

Case study: std::terminate

- Calling terminate has unspecified behaviour wrt stack unwinding
- 2008: The terminate() function shall not be called implicitly.
- 202x: Program-terminating functions should not be used
- Changes
  - We added explicit calls
  - We added more functions
  - We removed implicit calls => Other more specific rules
  - We added taking the address => Make it decidable
  - We really don’t mean those functions cannot be used!

What other changes did we do

- Update all rules to C++ 17
  - Each overriding virtual function shall be declared with the virtual keyword.
- Incorporate rule ideas from AUTOSAR and MISRA C
- Remove rules that we considered without real safety case
- Be more accurate in the rule amplification
- Rework/remove rules that were just not that good
  - Evaluation of the operand to the sizeof operator shall not contain side effects.
- Remove rules that were only requiring documentation
- …

What is coming next?

- MISRA C++ 202x
  - No date announced
  - Target: C++17
- Partial drafts should be available for review soon©
  - https://www.misra.org.uk/misra-cpp-202x-review-process/
- Tool vendors will provide updated tools
- We have no plans to stop after that
  - Some rules are omitted but could be added
  - A set of guidelines targeting parallel programming
  - Some guidelines about dynamic memory management?
  - C++20, C++23?