

C++ Core Guidelines Enforcement in Practice

PAUL JANSEN





Paul Jansen

- Graduated in philosophy and computer science
- Founder and CEO of TIOBE Software
- Company focus: measuring software quality
- Researching software quality: TIOBE Quality Indicator
- Checking more than 8,000 C/C++ industrial projects
- Researching programming languages: TIOBE Index



TIOBE Software

TIOBE Index

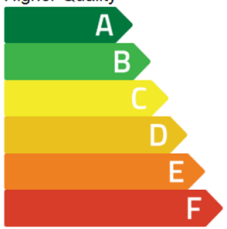
Sep 2026	Sep 2025	Change	Programming Language
1	1		Python
2	3	↑	C
3	2	↓	C++
4	4		Java
5	5		C#
6	6		JavaScript
7	7		Visual Basic
8	11	↑	SQL
9	13	⇈	R
10	18	⇈	Rust

- Popularity of 286 languages
- Based on search hits
- Updated every month
- 100,000+ visitors per month
- Heated debates
- C++ in top 4 for 35+ years
- C++ always been 5% - 15%

TIOBE Quality Indicator

aria2
src

Higher Quality



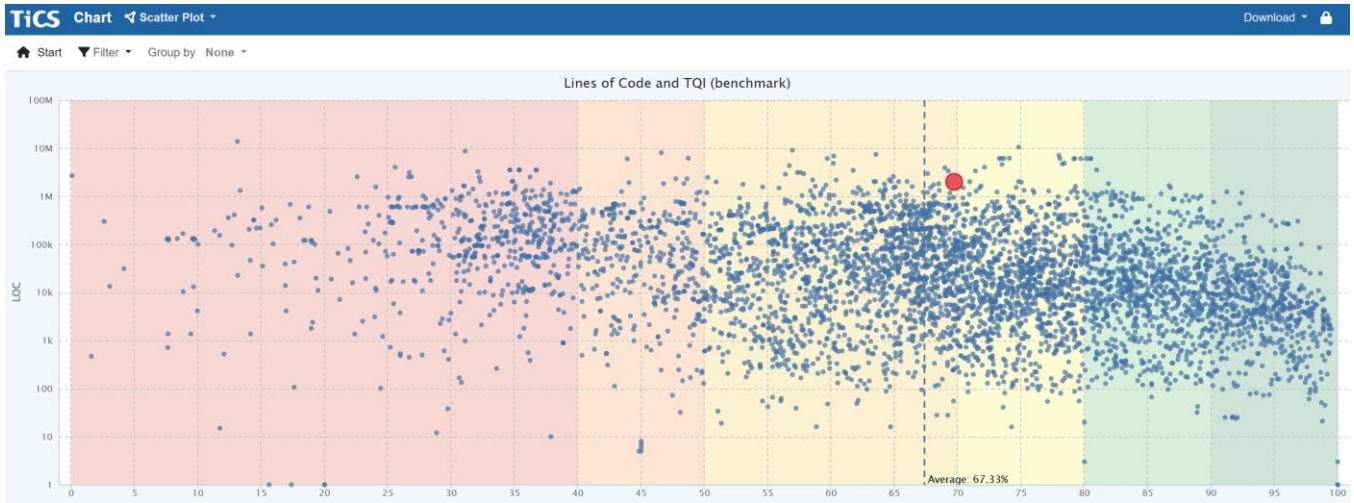
Lower Quality

TQI Score	71.61%	
Code Coverage	50.80%	D
Abstract Interpretation	86.28%	B
Code Security	55.53%	D
Compiler Warnings	45.04%	E
Cyclomatic Complexity	95.37%	A
Coding Standards	94.71%	A
Code Duplication	78.44%	C
Fan Out	60.77%	D

TiCS Measured on Jul 4, 2025

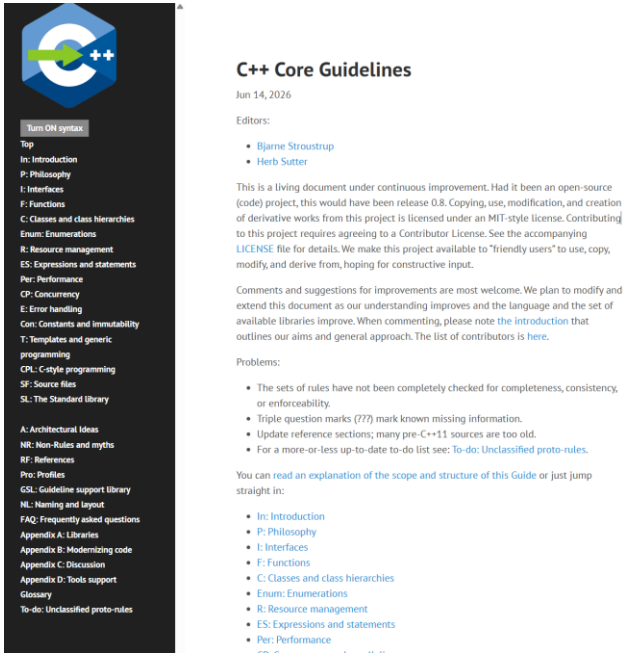
This product has been tested with the utmost care against the TIOBE Quality Indicator 4.17

- TIOBE Quality Indicator is based on ISO 25010 (software product quality)
- Below you see all 8,000+ projects: their sizes and TQI scores
- One of the metrics is Coding Standards, e.g. C++ Core Guidelines



What are the C++ Core Guidelines?

- The most comprehensive C++ coding standard
- It is about how to write modern C++ code
- Defined by Stroustrup and Sutter
- Open-source project, but moderated
- Updated every day
- isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines



How many rules are there in the C++ Core Guidelines?

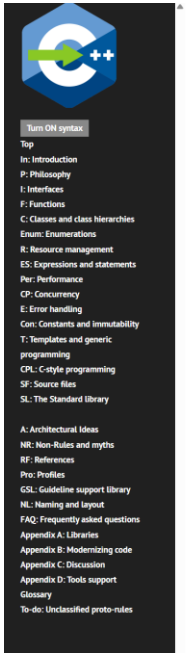
A. 42 rules

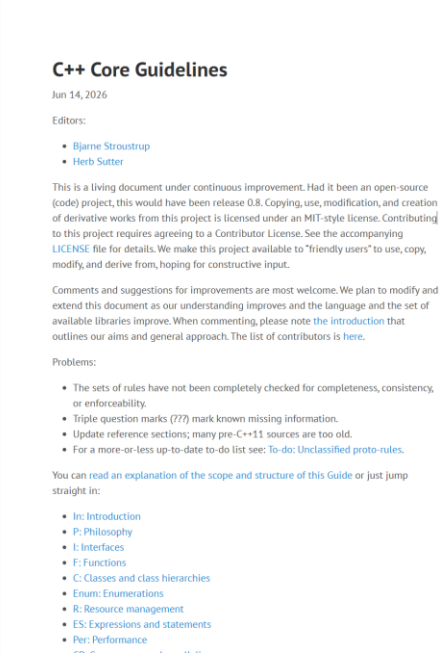
B. About 200 rules

C. 437 rules

D. More than 450 rules

E. Nobody knows





C++ Core Guidelines

Jun 14, 2026

Editors:

- Bjarne Stroustrup
- Herb Sutter

This is a living document under continuous improvement. Had it been an open-source (code) project, this would have been release 0.8. Copying, use, modification, and creation of derivative works from this project is licensed under an MIT-style license. Contributing to this project requires agreeing to a Contributor License. See the accompanying LICENSE file for details. We make this project available to "friendly users" to use, copy, modify, and derive from, hoping for constructive input.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve. When commenting, please note the introduction that outlines our aims and general approach. The list of contributors is here.

Problems:

- The sets of rules have not been completely checked for completeness, consistency, or enforceability.
- Triple question marks (???) mark known missing information.
- Update reference sections; many pre-C++11 sources are too old.
- For a more-or-less up-to-date to-do list see: To-do: Unclassified proto-rules.

You can read an explanation of the scope and structure of this Guide or just jump straight in:

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance

Enum.3: Prefer class enums over "plain" enums

Reason To minimize surprises: traditional enums convert to int too readily.

Example

```
void Print_color(int color);

enum Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;

// Clearly at least one of these calls is buggy.
Print_color(webby);
Print_color(Product_info::blue);
```

Instead use an `enum class`:

```
void Print_color(int color);

enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;
Print_color(webby);  // Error: cannot convert Web_color to int.
Print_color(Product_info::red);  // Error: cannot convert Product_info to int.
```

Enforcement (Simple) Warn on any non-class `enum` definition.

R.20: Use `unique_ptr` or `shared_ptr` to represent ownership

Reason They can prevent resource leaks.

Example Consider:

```
void f()
{
    X* p1 { new X };                // bad, p1 will leak
    auto p2 = make_unique<X>();     // good, unique ownership
    auto p3 = make_shared<X>();     // good, shared ownership
}
```

This will leak the object used to initialize `p1` (only).

Enforcement

- (Simple) Warn if the return value of `new` is assigned to a raw pointer.
- (Simple) Warn if the result of a function returning a raw owning pointer is assigned to a raw pointer.

C++ Core Guidelines in Industry

- Hardly anybody asks us to apply the C++ Core Guidelines…
- The (safety-critical) embedded industry wants/screams MISRA-C++ instead
- Why?
  - Demanded by their customers (check the box)
  - It is a well-defined set focusing on undefined behavior and type conversions
  - It is a checkable set

MISRA-C++

- Updates don't happen often
- "Only" undefined/unspecified behavior & type conversions
- "`unsigned int i = 1;`" is a violation...
- MISRA-C++ is an extended MISRA-C standard
- I think it is overhyped in safety-critical industries
- "No MISRA-C++ violations or no delivery"
- It is the only static check done...



Paul Jansen • You
Founder & CEO TIOBE Software (tiobe.com) | The Software Quality Company
1yr •

Dear European Automotive Industry, after having suffered yet another "MISRA experience", I felt it was time to write a letter. Let me know your thoughts!

TIOBE
the software quality company

TIOBE Software BV
Website: www.tiobe.com
Contact: info@tiobe.com
Phone: +31 (0)40 400 2800

Dear European Automotive Industry,

With this letter I would like to urge you to stop using the MISRA standard. The reason why I am doing this is because this standard is insufficiently focusing on improving the quality of automotive software. MISRA is very good in making sure C/C++ types are used correctly and unspecified behavior is identified. That's true. But it lacks any notion of what real good software code quality is about. MISRA is written to be checked by old-fashioned classic code checkers, but the world has moved on. We have deep flow checkers that can find real problems in software code, such as null pointer exceptions and array out of bounds. Moreover, the MISRA standard is revised only once in about 8 years, which is not acceptable in this fast-moving world.

Large car OEMs demand that their suppliers and the suppliers of their suppliers etc ad infinitum (we are talking about thousands of companies) adhere to the MISRA standard. This is done in such a way that any deviation, any violation that is not solved in the release, should be justified in writing. I have seen these suppliers creating 100,000+ reports just to prove that innocent conversions of types that are flagged by MISRA are justified, thus burning 100+ man-years every release to get their software accepted. If only MISRA was enforcing the right rules, the 100+ man-years could have actually helped your industry.

I have talked to these suppliers. They blindly follow what the OEMs have told them, and they even demand tool vendors to create MISRA code checkers. In other words, the specter of MISRA is haunting everyone in this industry. Once a supplier is MISRA compliant, they are satisfied because they can put a quality stamp on their release. They are not interested in delivering good software, they are interested in being compliant with "regulations". Please, stop with this bureaucracy and make sure real bugs are found in your code instead. Let's deliver good software and make sure your industry can grow instead of imploding due to the weight of its own bureaucratic procedures.

Yours sincerely,

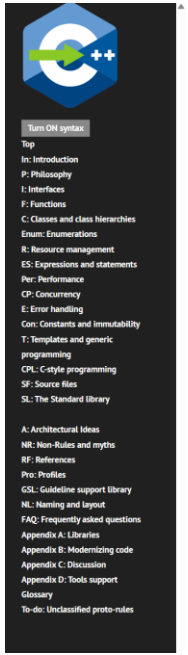
Paul Jansen
CEO TIOBE Software

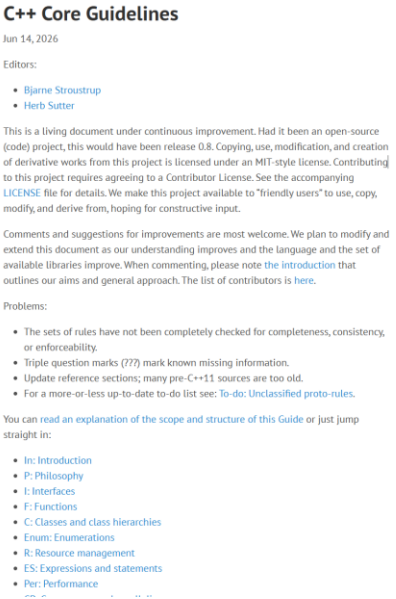
180 75 comments · 15 reposts

Why, oh why are the C++ Core Guidelines hardly used in industry?

C++ Core Guidelines: what is wrong?

- Weak versioning
- No severity levels (all rules are equal)
- Demand for GSL
- 268 occurrences of “???”
- Weak enforcements of rules (feels like an afterthought)
- Low applicability of some rules in industry





C++ Core Guidelines

Jun 14, 2026

Editors:

- Bjarne Stroustrup
- Herb Sutter

This is a living document under continuous improvement. Had it been an open-source (code) project, this would have been release 0.8. Copying, use, modification, and creation of derivative works from this project is licensed under an MIT-style license. Contributing to this project requires agreeing to a Contributor License. See the accompanying LICENSE file for details. We make this project available to "friendly users" to use, copy, modify, and derive from, hoping for constructive input.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve. When commenting, please note the introduction that outlines our aims and general approach. The list of contributors is here.

Problems:

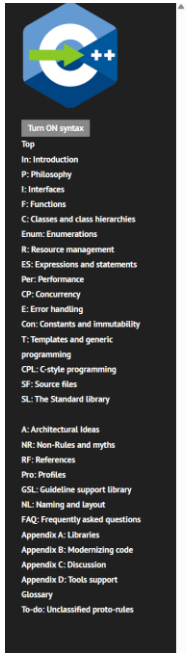
- The sets of rules have not been completely checked for completeness, consistency, or enforceability.
- Triple question marks (???) mark known missing information.
- Update reference sections; many pre-C++11 sources are too old.
- For a more-or-less up-to-date to-do list see: To-do: Unclassified proto-rules.

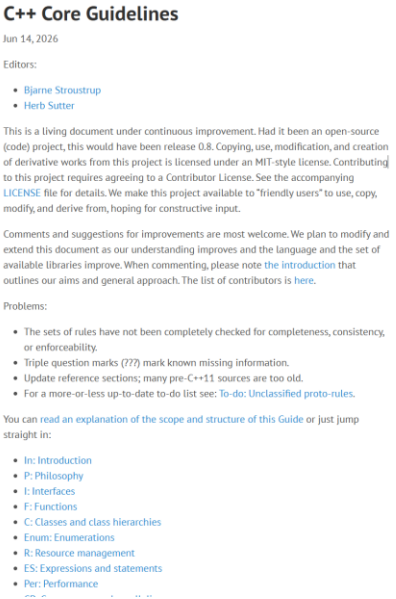
You can read an explanation of the scope and structure of this Guide or just jump straight in:

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance

C++ Core Guidelines: what is wrong?

- Weak versioning
- No severity levels (all rules are equal)
- Demand for GSL
- 268 occurrences of “???”
- **Weak enforcements of rules (feels like an afterthought)**
- Low applicability of some rules in industry





C++ Core Guidelines

Jun 14, 2026

Editors:

- Bjarne Stroustrup
- Herb Sutter

This is a living document under continuous improvement. Had it been an open-source (code) project, this would have been release 0.8. Copying, use, modification, and creation of derivative works from this project is licensed under an MIT-style license. Contributing to this project requires agreeing to a Contributor License. See the accompanying LICENSE file for details. We make this project available to "friendly users" to use, copy, modify, and derive from, hoping for constructive input.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve. When commenting, please note the introduction that outlines our aims and general approach. The list of contributors is here.

Problems:

- The sets of rules have not been completely checked for completeness, consistency, or enforceability.
- Triple question marks (???) mark known missing information.
- Update reference sections; many pre-C++11 sources are too old.
- For a more-or-less up-to-date to-do list see: To-do: Unclassified proto-rules.

You can read an explanation of the scope and structure of this Guide or just jump straight in:

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance

If a Coding Standard is not checked, it is not used!

- Even stronger: if the coding standard is not quality gated, it is not used!
- “Yes, we have a company coding standard!”
- “It is that document on the Wiki page…”
- Nobody knows all rules by heart
- Lots of discussion about the interpretation of rules, e.g. “use meaning full names”
- Most people don’t have the discipline to run the checker (hence gating is needed)
- **Hence, a clear rule enforcement is very important**

Examples Enforcements in the C++ Core Guidelines

- “Very hard in general.”
- “???” (many times)
- “(Not enforceable) This is a too complex pattern to reliably detect.”
- “Tricky. Requires semantic insight.”
- “Code review”
- “Hard. At best a heuristic. Look for an uninitialized variable followed by …”

Example Unrelated Enforcement C++ Core Guidelines

Enum.2: Use enumerations to represent sets of related named constants

Reason An enumeration shows the enumerators to be related and can be a named type.

• • •

Enforcement

- Flag `switch`-statements where the `case`s cover most but not all enumerators of an enumeration.
- Flag `switch`-statements where the `case`s cover a few enumerators of an enumeration, but there is no `default`.

Conversation 1 Commits 1 Checks 0 Files changed 1



simoes-tiobe commented on Sep 23 • edited

Summary

The title of C.45 reads:

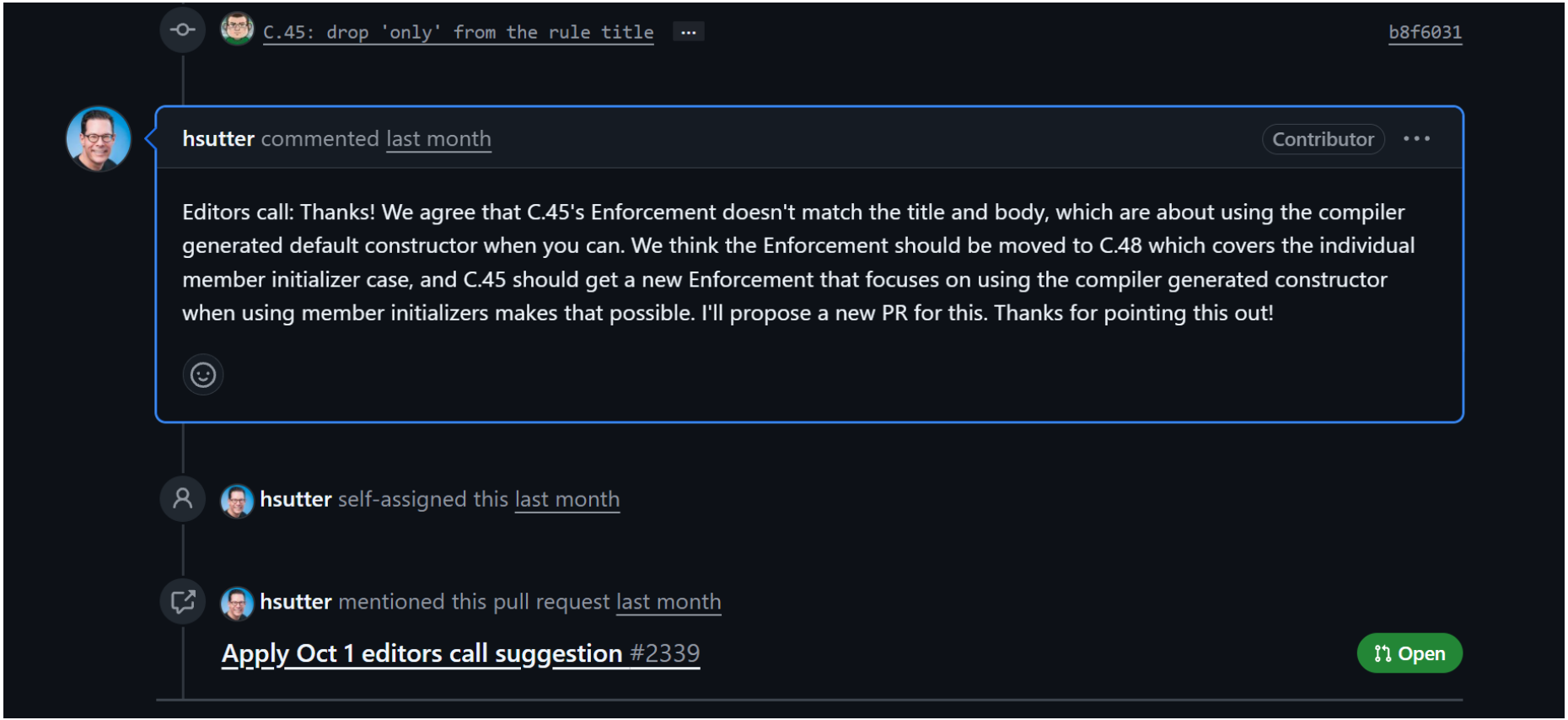
> Don't define a default constructor that **only** initializes data members

Taken literally, "only" scopes the rule to constructors whose entire body does nothing but initialize — i.e. it suggests the constructor is disqualified as soon as it does anything else, or has any initializer that can't become a default member initializer.

But the Enforcement clause is already narrower and per-initializer:

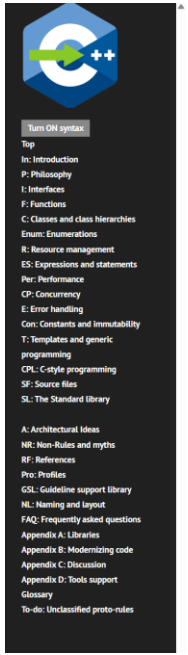
> (Simple) Flag if a default constructor's explicit member initializer is a constant, and recommend that the constant should be written as a data member initializer instead.

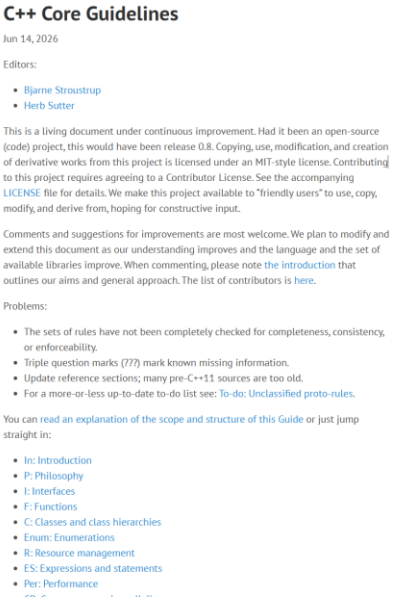
That is, each movable initializer should be flagged on its own merits, independent of the rest of the constructor. For example:


C.45: drop 'only' from the rule title
b8f6031
hsutter commented last month
Contributor
Editors call: Thanks! We agree that C.45's Enforcement doesn't match the title and body, which are about using the compiler generated default constructor when you can. We think the Enforcement should be moved to C.48 which covers the individual member initializer case, and C.45 should get a new Enforcement that focuses on using the compiler generated constructor when using member initializers makes that possible. I'll propose a new PR for this. Thanks for pointing this out!
hsutter self-assigned this last month
hsutter mentioned this pull request last month
Apply Oct 1 editors call suggestion #2339
Open

C++ Core Guidelines: what is wrong?

- Weak versioning
- No severity levels (all rules are equal)
- Demand for GSL
- 268 occurrences of “???”
- Weak enforcements of rules (feels like an afterthought)
- **Low applicability of some rules in industry**





C++ Core Guidelines

Jun 14, 2026

Editors:

- Bjarne Stroustrup
- Herb Sutter

This is a living document under continuous improvement. Had it been an open-source (code) project, this would have been release 0.8. Copying, use, modification, and creation of derivative works from this project is licensed under an MIT-style license. Contributing to this project requires agreeing to a Contributor License. See the accompanying LICENSE file for details. We make this project available to "friendly users" to use, copy, modify, and derive from, hoping for constructive input.

Comments and suggestions for improvements are most welcome. We plan to modify and extend this document as our understanding improves and the language and the set of available libraries improve. When commenting, please note the introduction that outlines our aims and general approach. The list of contributors is here.

Problems:

- The sets of rules have not been completely checked for completeness, consistency, or enforceability.
- Triple question marks (???) mark known missing information.
- Update reference sections; many pre-C++11 sources are too old.
- For a more-or-less up-to-date to-do list see: To-do: Unclassified proto-rules.

You can read an explanation of the scope and structure of this Guide or just jump straight in:

- In: Introduction
- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- Per: Performance

Industry needs more than a rule

- We run it for 100,000+ software engineers from all over the world
- It should work in all circumstances (because it is quality gated)
- No false positives, limited false negatives
- It should tackle the root cause, not a symptom
- Classic example: Yoda conditions:



```
if (5 == count)
```

C++ Core Guideline Rule I.24

I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning

- This won't work in practice. What about "void set_surface(int length, int width) { ... }"?
- How to fix this? Using different types? What is your idea?
- The real thing to be checked is if you swap arguments
- Hence, the implementation checks if you call it with "set_surface(width, length);"
- Yes, this might not flag everything, but what is flagged is probably a plain error

C++ Core Guideline Rule ES.45

ES.45: Avoid "magic constants"; use symbolic constants

Enforcement Flag literals in code. Give a pass to `0`, `1`, `nullptr`, `\n`, `""`, and others on a positive list.

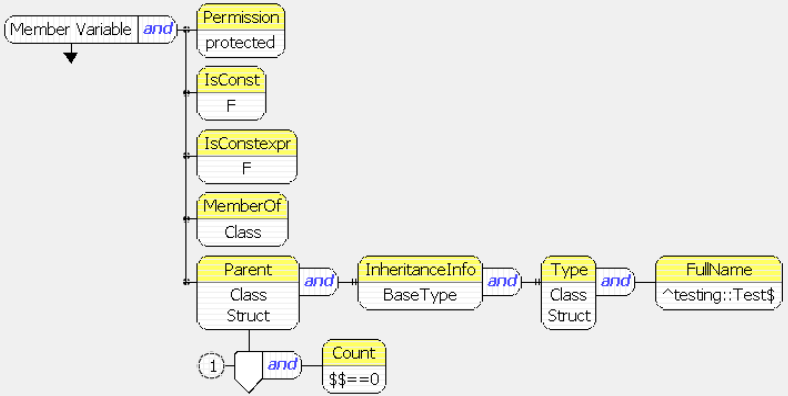
- What is "others"? We added factors of 2, factors of 10, 90/180/270/360 (for degrees), etc
- Still, lots of complaints. How to fix this? What is your idea?
- The real thing to be checked is if there are 2 magic numbers in the same scope
- Because if you change one of them, you probably want to change the other
- Yes, this might not flag everything, but what is flagged is an accident waiting to happen

C++ Core Guideline Rule C.37

C.37: Make destructors `noexcept`

- Wait, whut? Is my program supposed to crash if it throws something accidently?
- We are in a safety-critical program, remember?
- How to fix this? What is your idea?
- The real thing to be checked is whether you throw something from a destructor
- Hence, the implementation checks whether all function calls are “noexcept”
- Note that C functions that are called should be recognized, they are “noexcept” by default

- Implemented 110 rules so far
- Using Parasoft's RuleWizard
- Graphical parsing language
- Example to the right is C.133
- With an exception for Gtest...



A test fixture is created by inheriting from the ::testing::Test class and defining any required setup or teardown methods.

```
class CalculatorTest : public ::testing::Test {
protected:
    Calculator calc;

    void SetUp() override {
        // Runs before each test
    }

    void TearDown() override {
        // Runs after each test
    }
};
```

Conclusions

- The C++ Core Guidelines have a lot of potential
- It needs to be industrialized for adoption:
  - Add proper versioning
  - Add severity levels
  - Remove open ends
  - Professionalize enforcements
  - Sharpen some rules by means of a reality check (I am happy to help)
- Who wants to take up this challenge?

Which of the following languages is *not* in the TIOBE top 50?

A. Elixir

B. Zig

C. Lua

D. Rust

E. OCaml