MISRA C++:2023

Guidelines for the use of C++17 in critical systems

October 2023

Licensed to: gunjanjain
22 Apr 2025

First published October 2023 by The MISRA Consortium Limited
1 St James Court
Whitefriars
Norwich
Norfolk
NR3 1RU
UK

www.misra.org.uk

Copyright © 2023 The MISRA Consortium Limited

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise without the prior written permission of the Publisher.

"MISRA", "MISRA C" and the triangle logo are registered trademarks owned by The MISRA Consortium Limited.

Other product or brand names are trademarks or registered trademarks of their respective holders and no endorsement or recommendation of these products by MISRA is implied.

ISBN 978-1-911700-10-4 Paperback
ISBN 978-1-911700-11-1 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

License terms: this copy of MISRA C++:2023 is licensed to gunjanjain solely for your (or your employer's) usual business operations and your own personal use and study.

This document and any part of it, may not be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise without the prior written consent of MISRA. You are permitted to store a single additional copy solely for the purposes of backup.

For full details of the license terms please refer to the published terms of sale on the MISRA website. You agree to be bound by these license terms when using this document.

Licensed to: gunjanjain
22 Apr 2025

Foreword

Well, that was hard work, but it's finally done!

What started out as a "simple" revision to MISRA C++ to bring later versions of C++ into scope soon changed into a major update to ensure that, as much as possible, the guidance was in line with "modern C++ practices". A further consideration was the agreement with AUTOSAR to merge their C++14 guidelines (which in turn incorporated MISRA C++:2008) into the new version of MISRA C++.

The merger did increase the work that was needed, but it also helped as members of the AUTOSAR team joined the MISRA working group. This brought additional, experienced C++ practitioners on board, making it easier to obtain clarification where it was needed, and helped to ensure that any changes that were made were compatible with AUTOSAR's needs.

Of course, we also had to learn how to continue working during the global pandemic. Surprisingly, this didn't really have too much of an impact on our progress. In fact, it actually helped, as we soon found that well-structured, online meetings could be more effective as more people were able to attend — it's easier to take time out during the day to drop in and out of an "all-day" meeting than it is to spend additional time travelling to join a two-day face-to-face meeting.

The resulting document is a major step forward, but it is still very much a work in progress — C++ is a very big language, and there is still a lot of work to do. This document is only the first in a series of updates that will be used to increase coverage of the core language and the standard template library, and, when appropriate, to bring later versions of C++ into scope.

I would like to thank the Working Group for all of the time and effort they put into this document, and to congratulate them on how well they pulled together when the pandemic tried to get in the way.

Finally, I would like to give a special mention to my daughters Isobel and Jasmine for helping with some of the more tedious aspects of the final editorial work!

Chris Tapp, BSc (Hons) (Dunelm)
Chair, MISRA C++ Working Group
30th September 2023

Licensed to: gunjanjain
22 Apr 2025

MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

Disclaimer

Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and re-use.

Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations.

Licensed to: gunjanjain
22 Apr 2025

Acknowledgements

The MISRA Working Group

The MISRA Consortium would like to thank the following members of the MISRA C++ Working Group, and their employers, for their significant contribution to the writing of this document:

Jan Babst	Elektrobit Automotive GmbH
Xavier Bonaventura	BMW Group
Richard Corden	Perforce Software Inc
Patricia Hill	BUGSENG srl
Loïc Joly	SonarSource SA
Clive Pygott	LDRA Ltd (and Columbus Computing Ltd)
Michal Rozenau	Parasoft Corp.
Peter Sommerlad	Better Software, Wollerau, Switzerland
Stefan Staiger-Stöhr	Axivion GmbH (and Qt Group)
Chris Tapp	LDRA Ltd (and Keylevel Consultants Ltd)
Andreas Weis	Woven by Toyota, Inc.
Stephan Wilhelm	AbsInt Angewandte Informatik GmbH

Thanks also to the members of the MISRA C++ Working Group who supported our efforts:

Paul Anderson	GrammaTech, Inc.
John Bragg	MBDA UK Ltd
David Crocker	Escher Consulting Ltd
Prof. Dr. Jürgen Mottok	LaS^3, OTH Regensburg
Michael Wong	Codeplay Software

and the MISRA team who looked after us and kept us on track:

David Ward	HORIBA MIRA Ltd

AUTOSAR

The MISRA Consortium would like to thank AUTOSAR for offering us their guidelines and for supporting their merger into MISRA C++.

Our reviewers

Many thanks to the following individuals for the time and effort they put in during the review process:

Eng. Islam Fahd Ahmed	David Friberg	Kevin Leiffels	Andrew Scholan
Awais Arshad	Robert Gamble	Grant Lewis	Robert Seacord
Roberto Bagnara	Akrem Gassoumi	Thomas Loepfe	Robert Shade
Victor Bain	Thorsten Gecks	Tobias Loose	Jinlong Shen
Simone Ballarin	Masaki Gondo	Mikhail Maltsev	Elliot Simon
Verena Beckham	Guillaume Gonnet	Kyle Marcey	Saher Siwani

Licensed to: gunjanjain
22 Apr 2025

Oliver Beil
Freark Van Der Berg
Robert Bertossi
Arnab Bhaduri
Manuel Binna
Alexander Bock
Tan Li Boon
Simone Boscarato
Sam Bristow
John Brooks
Balazs Brosch
David Brown
Matthew Butler
Andriy Byzhynar
Nadege Caputo
David Chen
Weiren Chen
Mateusz Cholewiński
Robin Clay
Niall Cooling
Valery Creux
Kevin Dewald
Miodrag Djukic
Dariusz Donimirski
Fred Drury
Intars Dzalilovs
Andreas Fertig
Guillaume Granie
Charles-Henri Gros
Bruce Gu
Francisco Gutierrez
Paul Hampson
Jason Harper
Simon Hoinkis
Ralf Holly
Peter Holtwick
David Hooks
Ji Huang
Ichiro Inoue
Bastien Jauny
Seyeon Jeong
Kathleen Jones
Mischa Jonker
Richard Kaiser
Blaithin Kennedy
Thomas Kieß
Jan Koniarik
Leonidas Kosmidis
Sebastian Krings
Martin Krogmann
Minhyuk Kwon
Christopher Lapkowski
Peter Koch Larsen
David Ledger
Daniel Marjamäki
Christian Marx
Christof Meerwald
Andrew Emerson Meinke
Jan-Gerd Meß
Xiaoxiao Mo
Sebastian Moors
Gideon Müller
Yordan Naydenov
Jason Newell
Nhu Nguyen
Arthur O'Dwyer
Guido Persch
Jonas Persson
Tobias Pfeffer
Dr. Michael Pfeiffer
Heiko Poelstra
Sreeram K R
Johan Regin
Dr. Paul Reitz
Omar Rekik
Derek Repsch
Clemens Richter
Roland A.I. Rosier
Tsuyoshi Sakurai
José Daniel Garcia Sanchez
Jan Sommer
Philipp Sommer
Hogil Song
Nicolai Spohrer
Michał Staroń
Michael Stevens
Hashimoto Takahiro
Yasushi Tanaka
Piotr Tański
Felix Tarköy
Jan Toennemann
Drago Trusk
Vijaykumar Vaghasiya
Jussi Vänskä
John Waddle
Liz Whiting
Martin Willers
Daniel Withopf
Ian Wright
Yohei Yamaguchi
Naoki Yoshikara
Mark Allan Young
Achim Olaf Zacher
Vyacheslav Zavadsky
Ivan Zhdanov
Christian Zimmermann
Wojciech Zimny

Other acknowledgements

DokuWiki was used extensively during the drafting of this document. Our thanks go to all those involved in its development.

This document was typeset using fonts licensed under the SIL Open Font License, Version 1.1:

- Open Sans — Copyright 2020, The Open Sans Project Authors
- Fira Code — Copyright 2014–2020, The Fira Code Project Authors

Licensed to: gunjanjain
22 Apr 2025

Contents

1 Introduction 1
- 1.1 Background 1
- 1.2 The vision 1
- 1.3 Scope 1
- 1.4 Adoption 2
- 1.5 Compliance 2

2 Background to MISRA C++ 3
- 2.1 Popularity 3
- 2.2 Disadvantages of C++ 3
- 2.3 The use of C++ in critical systems 4

3 Introduction to the guidelines 5
- 3.1 Guideline classification 5
- 3.2 Guideline categories 5
- 3.3 Organization of guidelines 6
- 3.4 Redundancy in the guidelines 6
- 3.5 Decidability of rules 6
- 3.6 Analysis scope 7
- 3.7 Applicability 8
- 3.8 Presentation of guidelines 9
- 3.9 Understanding the source references 10

4 Guidelines 12
- 4.0 Language independent issues 12
- 4.4 General principles 29
- 4.5 Lexical conventions 33
- 4.6 Basic concepts 43
- 4.7 Standard conversions 70
- 4.8 Expressions 87
- 4.9 Statements 113
- 4.10 Declarations 127
- 4.11 Declarators 135
- 4.12 Classes 141
- 4.13 Derived classes 144
- 4.14 Member access control 150
- 4.15 Special member functions 151
- 4.16 Overloading 166
- 4.17 Templates 169

Licensed to: gunjanjain
22 Apr 2025

4.18 Exception handling 170
4.19 Preprocessing directives 181
4.21 Language support library 194
4.22 Diagnostics library 206
4.23 General utilities library 208
4.24 Strings library 210
4.25 Localization library 212
4.26 Containers library 215
4.28 Algorithms library 216
4.30 Input/output library 221

5 References 224
5.0 MISRA publications 224
5.1 International standards 224
5.2 Other references 225

Appendix A Summary of guidelines 226

Appendix B Guideline attributes 238

Appendix C Glossary 243

Licensed to: gunjanjain
22 Apr 2025

1 Introduction

1.1 Background

The MISRA C++ Guidelines define a subset of the C++ language in which the opportunity to make mistakes is either removed or reduced. Many standards for the development of safety-related software require, or recommend, the use of a language subset, and this can also be used to develop any application with security, high integrity or high reliability requirements.

As well as defining this subset, these MISRA C++ Guidelines will:

- Provide educational material for those developing C++ programs;
- Provide reference material for tool developers.

1.2 The vision

It is a long-term objective for MISRA C++ to define a "predictable subset" of the C++ language, and to provide explicit guidance for the avoidance of all instances of undefined and unspecified behaviour. For this reason, features that are initially permitted without the support of specific guidelines may be subject to restrictions in the future.

The vision for MISRA C++ is to enhance the existing guidance by:

- Tracking the evolution of the C++ language;
- Correcting any known issues with the previous editions;
- Adding new guidelines for which there is a strong rationale;
- Removing any guidelines for which the rationale is insufficient;
- Improving the specification and the rationale for existing guidelines, where appropriate;
- Increasing the number of guidelines that can be processed by static analysis tools by improving their decidability, where possible.

1.3 Scope

This document supports the use of ISO/IEC 14882:2017 [8], commonly referred to as "C++17", within a project. The term *the C++ Standard* is used within the text to refer to ISO/IEC 14882:2017.

Notes:

1. Access to a copy of the relevant C++ Standard is not necessary for the use of MISRA C++, but it may be helpful.
2. The edition of the C++ Standard chosen for use within a project may be influenced by factors such as the amount of legacy code being reused within a project and/or the availability of compilers for the target processor.
3. Use of this document to support code developed to a different version of C++ requires a thorough investigation to be carried out to understand and document the differences in behaviour between it and C++17. The guidelines within this document target C++17, and they may not be appropriate or adequate when applied to other versions of the language — specifically, additional guidelines may be needed to prevent undesirable behaviours that are not covered by this document.

Licensed to: gunjanjain
22 Apr 2025

1.4 Adoption

MISRA C++ should be adopted at the outset of a project.

Note: if a project is building on existing code that has a proven track record, then the benefits of compliance with MISRA C++ may be outweighed by the risks of introducing defects when making the code compliant. In such cases, a judgement should be made between the benefits of adoption and the risks of introducing defects.

1.5 Compliance

The guidance given in the current edition of MISRA Compliance [1] shall be followed when making a claim of compliance with MISRA C++.

Licensed to: gunjanjain
22 Apr 2025

2 Background to MISRA C++

2.1 Popularity

The C++ programming language [8] is now widely used for critical systems, due largely to the inherent language flexibility, the extent of support and its potential for portability across a wide range of hardware. Specific reasons for its use include:

- Compilers are readily available for many processors;
- C++ programs can be compiled to efficient machine code;
- C++ enables object-oriented design methods to be used;
- It is defined by an international standard;
- It provides mechanisms to access the input/output capabilities of the target processor, whether directly or by means of language extensions;
- There is a considerable body of experience with using C++ in critical systems;
- It is widely supported by static analysis and test tools.

2.2 Disadvantages of C++

While popular, no programming language can guarantee that the final executable code will behave exactly as the developer intended. The language has several drawbacks which are discussed in the following sub-sections.

2.2.1 Language definition

The C++ Standard does not specify the language completely but places some aspects under the control of an implementation. This is intentional, partly because of the desire to support many pre-existing implementations for widely different target processors. As a result, there are areas of the language in which:

- The behaviour is undefined;
- The behaviour is unspecified;
- An implementation is free to choose its own behaviour provided that it is documented.

A program that relies on undefined or unspecified behaviour is not necessarily guaranteed to behave in a predictable manner.

A program that places excessive reliance on implementation-defined behaviour may be difficult to port to a different target. The presence of implementation-defined behaviour may also hinder static analysis if it is not possible to configure the analyser to handle it.

Note: unfortunately, unlike for C, the C++ Standard does not enumerate these behaviours.

2.2.2 Language misuse

While C++ programs can be laid out in a structured and comprehensible manner, C++ makes it easy for developers to write obscure code that is difficult to understand.

The specification of the operators makes it difficult for programming errors to be detected by a compiler. For example, the following two fragments of code are both perfectly legal and it is impossible for a compiler to know whether one has been mistakenly used in place of the other:

```
if ( a == b )   // Tests whether a and b are equal
if ( a =  b )   // Assigns b to a and tests whether a is non-zero
```


Licensed to: gunjanjain
22 Apr 2025

Aspects of type checking for the *fundamental types* within C++ are weak. For example, the language will allow a floating-point value to be stored in an object of type `bool`. This, and other, type mismatches are not required to be diagnosed by the compiler — implicit type conversions are introduced to allow the code to execute in the way that has been expressed by the developer. These conversions are benign in some cases, but others can lead to faults.

2.2.3 Language misunderstanding

There are areas of the language that are commonly misunderstood by developers. For example, C++ has more operators than some other languages and consequently has a high number of different operator precedence levels, some of which are not intuitive.

The implicit type conversion provided by C++ can also be confusing to developers who are familiar with strongly-typed languages. For example, operands may be "promoted" to wider types, meaning that the type resulting from an operation is not necessarily the same as that of the operands.

2.2.4 Run-time error checking

C++ programs can be compiled into small and efficient machine code, but the trade-off is that there is a very limited degree of run-time checking. C++ programs generally do not provide run-time checking for common problems such as arithmetic exceptions (e.g. divide by zero), overflow, validity of pointers, or array bound errors. The C++ philosophy is that the developer is responsible for making such checks explicitly.

2.3 The use of C++ in critical systems

The C++ programming language is commonly selected for software in critical systems, whether safety- or security-related, for both freestanding (embedded) and hosted applications.

The recommendations within these Guidelines, when used within a documented software development process, address many of the disadvantages of the C++ language, irrespective of the purpose of the end-use of the developed code.

These Guidelines are therefore equally as applicable within a security-related environment as they are within a safety-related one.

Licensed to: gunjanjain
22 Apr 2025

3 Introduction to the guidelines

This section explains the presentation of the guidelines in Section 4, the main content of this document, and serves as an introduction to the content of that section.

3.1 Guideline classification

Every MISRA C++ guideline is classified as either being a "directive" or a "rule".

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 different 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 subject to the limitations described in Section 3.5.

The numbering for each guideline includes either "Dir" or "Rule" as a prefix, as appropriate to its classification.

3.2 Guideline categories

Every guideline within this document is given a single category of "mandatory", "required" or "advisory". In addition, any "advisory" guideline may also be re-categorized as "disapplied", as described within MISRA Compliance [1]. The meanings of these categories are described below.

Note: this document does not give, nor intend to imply, any grading of importance for guidelines having the same category. All required guidelines, whether directives or rules, should be considered to be of equal importance, as should all mandatory and advisory ones.

3.2.1 Mandatory guidelines

Code which is claimed to conform to this document shall comply with every mandatory guideline — violations of mandatory guidelines are never permitted, and they can never be subject to deviation.

3.2.2 Required guidelines

Code which is claimed to conform to this document shall either comply with every required guideline, or formal deviations shall be produced to justify any violations. The deviation process is described within MISRA Compliance [1].

An organization or project may choose to re-categorize any required guideline as if it were mandatory.

3.2.3 Advisory guidelines

Advisory guidelines contain recommendations that help to improve a project's code quality attributes. However, the status of advisory does not mean that these items can be ignored, but rather that they should be followed as far as is reasonably practical.

Code which is claimed to conform to this document shall either comply with every advisory guideline, or documentation shall be produced to justify any violations — formal deviations are not necessary, but alternative arrangements should be made to document and justifying violations if the formal deviation process is not applied.

Licensed to: gunjanjain
22 Apr 2025

An organization or project may choose to re-categorize any advisory guideline as if it were mandatory, required or disapplied (subject to the requirements of MISRA Compliance [1]).

3.2.4 Disapplied guidelines

Code which is claimed to conform to this document is not required to enforce any disapplied guideline — violations may simply be disregarded, with documentation being produced to justify the disapplication of the guideline.

Note: none of the guidelines within this document are categorized as disapplied, but advisory guidelines may be re-categorized as disapplied.

3.3 Organization of guidelines

The guidelines are organized using the same structure as the sections of the C++ Standard. However, there is inevitably overlap, with one guideline possibly being relevant to a number of sections. Where this is the case, the guideline has been placed in the most relevant section.

3.4 Redundancy in the guidelines

There are a few cases within this document where a guideline is given that refers to a language feature that is banned or advised against elsewhere in the document. This is intentional. It may be that the developer chooses to use that feature, either by raising a deviation against a required guideline, or by choosing not to follow an advisory guideline. In this case the second guideline, constraining the use of that feature, becomes relevant.

3.5 Decidability of rules

Each mandatory, required and advisory rule is classified as *decidable* or *undecidable*. This classification describes the theoretical ability of a static analyser to answer the question "Does this code comply with this rule?" The directives are **not** classified in this way because it is impossible, given only the source code, to devise an algorithm that could guarantee to check for compliance.

A rule is *decidable* if it is possible for a program to answer the question with a "yes" or a "no" **in every case** and *undecidable* otherwise. A review of the theory of computation, on which this classification is based, is beyond the scope of this document but a rule is likely to be undecidable if detecting violations depends on run-time properties such as:

- The value that an object holds;
- Whether control reaches a particular point in the program.

Decidable rules have useful properties with regard to static analysis. Provided that a defect-free and complete static analyser is configured correctly:

- A reported violation of a decidable rule indicates a real violation;
- No reported violation of a decidable rule indicates there are no violations in the code being analysed.

Some examples of decidable rules are:

- Rule 6.4.1 — depends on the names and scopes of identifiers;
- Rule 8.2.7 — depends on the type being cast;
- Rule 19.3.4 — depends on the result of a macro expansion.

Licensed to: gunjanjain
22 Apr 2025

Static analysers vary in their ability to detect violations of undecidable rules:

- A reported violation of an undecidable rule may not necessarily indicate a real violation; some analysers take the approach of reporting **possible** violations to remind users of the uncertainty;
- No reported violation of an undecidable rule does not necessarily indicate that there are no violations in the code being analysed.

Note: if a tool produces a diagnostic for a violation of an undecidable guideline (regardless of category), and a justification can be provided to show that a violation cannot occur at run time, then the diagnostic does not actually indicate a violation. In that case, the justification should be recorded to support the claim that there is no violation of a MISRA guideline. See MISRA Compliance [1].

Some examples of undecidable rules are:

- Rule 8.2.10 — depends on knowing which functions are called when function pointers are used;
- Rule 11.6.2 — depends on data-dependent control flow.

As indicated in MISRA Compliance [1], a process should be developed for analysing the results of static analysis and recording the outcome. Particular attention should be paid to the process for analysing any output that relates to undecidable rules.

3.6 Analysis scope

Each rule is classified according to the amount of code that needs to be checked in order to detect violations. As for decidability, the concept of analysis scope is not applied to directives.

The analysis scopes that may be applied to rules are "Single Translation Unit" and "System".

If a rule is classified as capable of being checked on a "Single Translation Unit" basis then it is possible to detect all violations within a project by checking each *translation unit* independently. For example, the presence of *switch* statements that do not contain *default* labels (Rule 9.4.2) within one *translation unit* has no effect on whether other *translation units* contain such *switch* statements.

If a rule is classified as needing to be checked on a "System" basis then identifying violations of a rule within a *translation unit* requires checking more than the *translation unit* in question. Rules that are classified as "System" are best checked by analysing all the source code, although it will be possible to identify some violations when checking a subset of the whole source. For example, if a project has two *translation units A* and *B*, it is possible to check that all declarations of a function within each *translation unit* have compatible types (Rule 6.2.2). However, this does not guarantee that the declarations in *A* are compatible with those in *B*. All of the source code that will be compiled and linked into the executable therefore needs to be checked to guarantee compliance with this rule.

Most undecidable rules need to be checked on a "System" basis because, in the general case, information about the behaviour of other *translation units* will be needed. For example, whether or not the value of the automatic object `x` is set before `x` is used (Rule 11.6.2) in the function `g`, below, will depend on the behaviour of the function `f` which is defined in another *translation unit*:

```
extern void f ( uint16_t * p );

uint16_t  y;

void g ( void )
{
  uint16_t  x;  // x is not given a value

  f ( &x );     // f might modify the object pointed to by its parameter
  y = x;        // x may or may not be unset
}
```


Licensed to: gunjanjain
22 Apr 2025

3.7 Applicability

3.7.1 Conditional compilation

Unless otherwise stated, the guidelines shall only be applied to code that remains after preprocessing of a *translation unit*.

Note: this requires that the values of all macros used to control conditional compilation be made available to an analysis tool, including those that are defined outside of the source code (e.g. those defined by means of a command line parameter, or that are defined by the compiler).

3.7.2 Classes

Unless otherwise stated, any reference to "a class" within the guidelines refers to any type introduced with the `class` or `struct` keyword. Note that the C++ Standard also includes types introduced with the `union` keyword when referring to class types, but that is not the case within the guidelines.

3.7.3 Templates

Unless otherwise stated, the guidelines shall only be applied to fully instantiated templates.

3.7.4 Compiler generated code

Unless otherwise specified, all guidelines shall apply to implicitly-declared or implicitly-defined special member functions.

3.7.5 Automatically generated code

The MISRA C++ guidelines are applicable to code that has been generated automatically. Responsibility for compliance lies both with the developer of the automatic code generation tool, and with the developer of the model from which code is being generated. Since there are several modelling packages, each of which may have several automatic code generators, it is not possible to allocate this responsibility individually for each MISRA C++ guideline. It is expected that users of modelling packages and code generators will employ relevant guidelines such as MISRA AC GMG [4] and MISRA AC SLSF [5].

Note: the guidance given in the current edition of MISRA Compliance [1] shall be followed when making a claim that automatically generated code is compliant with MISRA C++.

Licensed to: gunjanjain
22 Apr 2025

3.8 Presentation of guidelines

The guidelines within this document are presented in a number of sections, with the headings of those sections tracking the headings of the sections within the C++ Standard that cover the language behaviours to which the guidelines they contain are related. The headings also include the *stable names* from the C++ Standard that allow content to be located even if the sections are renumbered (e.g. between versions). Sections with the `[misra]` *stable name* are an exception, and are used to identify sections of best-practice guidance that is not directly attributable to language behaviours.

Within the sections, the individual guidelines have the following format:

Ident	Headline

[Source ref]

Category Category

Analysis Decidability, Scope

where:

- **Ident** is a unique identifier for the guideline of the form **prefix a.b.c**, where:
  - **prefix** is either "Dir" or "Rule", as appropriate to the guideline's classification;
  - **a.b** reflects the section number within the C++ Standard to which the guideline is related;
  - **c** is a sequence number for guidelines related to the above section;
- **Headline** is a summary of the guideline;
- **Source ref** indicates the primary source(s) which led to this item or group of items, where applicable, as explained in Section 3.9;
- **Category** is one of "Mandatory", "Required" or "Advisory", as explained in Section 3.2;
- **Decidability** is one of "Decidable" or "Undecidable", as explained in Section 3.5;
- **Scope** is one of "System" or "Single Translation Unit", as explained in Section 3.6;

Notes:

1. As guidance is not given for every section within the C++ Standard, the numbering of the designators **a.b** and the section headings within this document are not contiguous;
2. Section **0.b** contains general guidance that does not relate to any specific area within the C++ Standard;
3. Sections of the form **a.0** give guidance that relates to section **a** of the C++ Standard, but which is not directly attributable to any particular language construct within that section.
4. The "Analysis" line is omitted from directives as decidability and analysis scope do not apply.

In addition, supporting text is provided for each item or group of related items. The text gives, where appropriate, some explanation of the underlying issues being addressed by the guideline(s), and examples of how to apply them.

Within the supporting text, there may be a heading titled "Amplification", followed by text that provides a more precise description of the guideline. An amplification is normative; where its requirements differ from those of the headline, the amplification takes precedence. This mechanism is convenient as it allows a complicated concept to be conveyed using a short headline.

Within the supporting text, there may be a heading titled "Exception", followed by text that describes situations in which the guideline does not apply. The use of exceptions permits the description of some guidelines to be simplified. It is important to note that an exception is a situation in which the guideline does not apply. Code that complies with a guideline by virtue of an exception does not require a deviation.

Licensed to: gunjanjain
22 Apr 2025

The supporting text is not intended as a tutorial in the relevant language feature, as the reader is assumed to have a working knowledge of the language. Further information on the language features can be obtained by consulting the relevant section of the C++ Standard or other C++ language references. Where a source reference is given for one or more of the undesirable behaviours listed in the C++ Standard, then the original issue raised in it may provide additional help in understanding the guideline.

Within the guidelines, and their supporting text, the following font styles are used to represent keywords and code:

- Items defined in the C++ Standard or the Glossary appear in *italic text*;
- Keywords and language constructs appear in a `monospaced font`;
- Code also appears in a `monospaced font`, either within other text; or

```
As separate code fragments
```

Within the code examples:

- For the sake of brevity, code fragments may be incomplete (for example, implicit use of Standard Library header files, `if` statements without a body).
- Use is made of the fixed-width integer types from `<cstdint>`.
- Where an object's declaration is omitted, its type can be deduced from its name, with the introductory characters indicating the type:
  - A name starting with `u8`, `s8`, `u16`, etc. has type `uint8_t`, `int8_t`, `uint16_t`, etc.
  - A name starting with `f`, `d` or `ld` has type `float`, `double` or `long double`.

3.9 Understanding the source references

3.9.1 References to the C++ Standard

References into the C++ Standard are used to identify vulnerabilities which need to be mitigated:

- *Undefined behaviour* — Behaviour for which the C++ Standard imposes no requirements (see [defns.undefined]). These are essentially programming errors for which the compiler is not obliged to issue a diagnostic (error message), and are particularly important from a safety point of view, as they represent programming errors which may not be diagnosed by the compiler. For example, Rule 6.8.1 covers object lifetime violations.
- *Unspecified behaviour* — Behaviour within the C++ Standard that depends on the implementation (see [defns.unspecified]). These are language constructs that must compile successfully, but the implementation may choose from a set of appropriately defined behaviours. For example, Rule 8.2.6 covers casting to pointer types.
- *Implementation-defined behaviour* — Behaviour within the C++ Standard that depends on, and is documented by, the implementation (see [defns.impl.defined]). These are similar to *unspecified behaviour*, except that the behaviour must be consistent and documented. However, the related behaviour can vary from one compiler to another. For example, Rule 6.9.2 covers the sizes of the integral types.
- *Conditionally-supported behaviour* — A program construct defined within the C++ Standard that an implementation is not required to support (see [defns.conf.supp]). These are language constructs that may not be provided by all compilers. An implementation is required to document those features that are not supported and to issue a diagnostic if an unsupported construct is used within the code. For example, Rule 10.4.1 covers the use of `asm`.
- *No diagnostic required* — Behaviour within the C++ Standard for which no diagnostic is required (see [intro.compliance]/1). These are conditions that may lead to program errors, but for which

Licensed to: gunjanjain
22 Apr 2025

the C++ Standard explicitly states that the implementation is not required to issue a diagnostic (error message). For example, Rule 6.2.1 covers the *one-definition rule*.

A guideline providing mitigation for such a vulnerability will include a reference into the C++ Standard having the form:

```
[Stable-name] Vulnerability Paragraph
```

where:

- **Stable-name** is an identifier used within the C++ Standard to identify a specific section.
- **Vulnerability** is as indicated above.
- **Paragraph** identifies the paragraph within the section where the vulnerability occurs.

For example, a reference of:

```
[intro.execution] Undefined 17
```

would indicate that a guideline targets the undefined behaviour found in paragraph 17 of the [intro.execution] section of the C++ Standard.

In addition, supporting context for a guideline's rationale will use a reference having the form:

```
[Stable-name] / Paragraph
```

3.9.2 Other references

References to other sources may be consulted by a reader wishing to gain a fuller understanding of the rationale behind a guideline (for example when considering a request for a deviation). These references have the form:

```
[Reference] Location
```

where:

- **Reference** identifies the referenced material.
- **Location** identifies a page, section or paragraph within the document.

The following source references are used:

Reference	Source
C11	ISO/IEC 9899:2011 [6]
ISO/IEC/IEEE 60559	ISO/IEC/IEEE 60559:2011 [10]
IEC 61508	IEC 61508 [11]
ISO 26262	ISO 26262 [9]
DO-178C	DO-178C [12]
MISRA Guidelines	MISRA Development guidelines for vehicle based software [3]
Koenig	C Traps and Pitfalls [14]

Licensed to: gunjanjain
22 Apr 2025

4 Guidelines

4.0 Language independent issues

4.0.0 Path feasibility [misra]

Rule 0.0.1 A function shall not contain *unreachable* statements

[IEC 61508-7] / C.5.9
[DO-178C] / 6.4.4.3.c
[ISO 26262-6] / 9.4

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A statement is *unreachable* if the block containing it is not *reachable* from the *entry block* of the Control Flow Graph (CFG) for the function.

For the purpose of this rule:

- Both operands of a *reachable* logical AND (`&&`) or logical OR (`||`) operator are considered *reachable*; and
- All three operands of a *reachable* conditional operator (`?:`) are considered *reachable*; and
- The blocks linked by the edges from a *condition* of a *selection-statement* or an *iteration-statement* are all considered *reachable*, except when the *condition* is a *constant expression*, in which case only the blocks linked by edges selected by the *condition* are considered *reachable*; and
- A call to a function declared `[[noreturn]]` has no CFG out edge; and
- If a `try` *compound-statement* of a *(function-)try-block* does not contain a *reachable*, *potentially-throwing* statement, then all *catch-handlers* are *unreachable*, otherwise all *catch-handlers* are considered *reachable* subject to the restriction that a *catch-handler* that appears after a more generic *handler* of the same *try-block* is not *reachable*.

The rule does not apply to *statements* in the discarded branch of a `constexpr if` *statement*.

Rationale

Unreachable code often indicates a defect in the program, as, assuming that the program does not exhibit any *undefined behaviour*, *unreachable* code cannot be executed and cannot have any effect on the program's outputs.

In order to avoid crosstalk with Rule 0.0.2, the handling of logical and conditional operators in the conceptual CFG used by this rule differs from that in a traditional CFG.

Licensed to: gunjanjain
22 Apr 2025

Example

```
bool f0();

int32_t f1( int32_t c, int32_t & res )
{
  if ( false && f0() ) { }  // Compliant - statement is considered to be reachable

  return res;

  res = c;                  // Non-compliant - not reachable from entry block

  bool result;              // Non-compliant - not reachable from entry block
}

void f2( int32_t i )
{
  while ( true )            // Constant condition - single edge into body of loop
  {
    if ( i != 0 )
    {
      break;                // Adds edge to statements following the loop body
    }
  }

  ++i;                      // Compliant - reachable via 'break'

  while ( true )            // Constant condition - single edge into body of loop
  {
    f();
  }

  ++i;                      // Non-compliant - not reachable from entry block
}

void f3( int32_t i )
{
  goto LABEL;
  ++i;                      // Non-compliant - no edge to this block

LABEL:
  ++i;                      // Compliant
}

class BaseException {};
class DerivedException: public BaseException {};

void f4()
{
  try { /* ... */ }
  catch ( BaseException    & b ) { }
  catch ( DerivedException & d ) { }   // Non-compliant - will be caught above
}

void f5() noexcept;

void f6()
{
  try { f5(); }
  catch ( int32_t ) { }     // Non-compliant - f5 is not potentially-throwing
  catch ( ... )     { }     // Non-compliant - f5 is not potentially-throwing
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f7( int32_t i )
{
  try
  {
    throw i;
    ++i;                    // Non-compliant - no edge to this block
  }
  catch ( int32_t ) { }     // Compliant - all catch-handlers are reachable
  catch ( int16_t ) { }     // Compliant - all catch-handlers are reachable

  ++i;                      // Compliant
}

void f8();

int32_t f9( int32_t i )
{
  try
  {
    f8();                   // Potentially-throwing
    return i * 2;           // Compliant
  }
  catch( int32_t ) { }      // Compliant - all catch-handlers are reachable

  return 0;                 // Compliant - even if f8 throws a type
}                           //             other than int32_t

[[noreturn]] void f10() noexcept;

int32_t f11()
{
  f10();                    // Does not return
  return 0;                 // Non-compliant
}
```

Rule 0.0.2 Controlling expressions should not be invariant

[IEC 61508-7] / C.5.9
[DO-178C] / 6.4.4.3.c
[ISO 26262-6] / 9.4.5

Category Advisory

Analysis Undecidable, System

Amplification

This rule applies to:

- Controlling expressions of `if`, `while`, `for`, `do ... while` and `switch` statements; and
- The first operand of the conditional operator (`?:`); and
- The left hand operand of the logical AND (`&&`) and logical OR (`||`) operators.

It does not apply to controlling expressions of `constexpr if` statements.

A function's compliance with this rule is determined independently of the context in which the function is called. For example, a Boolean parameter is treated as if it may have a value of `true` or `false`, even if all the calls expressed in the current program use a value of `true`.

Licensed to: gunjanjain
22 Apr 2025

Rationale

If a controlling expression has an invariant value, it is possible that there is a programming error. Any code in an *infeasible path* may be removed by the compiler, which might have the effect of removing code that has been introduced for defensive purposes.

This rule does not apply to `constexpr if`, as this is intended to be evaluated at compile time and requires a constant expression.

Exception

1. A `while` statement with a constant expression evaluating to `true` is permitted as this is commonly used in real time systems.
2. Macros are permitted to expand to a *do-while* statement of the form `do { } while ( false )`, allowing a macro expansion to be used as a statement that includes a local scope.

Example

```
s8a = ( u16a < 0u ) ? 0 : 1;                // Non-compliant - u16a always >= 0

if ( u16a <= 0xffffu ) { }                  // Non-compliant - always true
if ( 2 > 3 ) { }                            // Non-compliant - always false
if ( ( s8a < 10 ) && ( s8a > 20 ) ) { }     // Non-compliant - always false
if ( ( s8a < 10 ) || ( s8a > 5  ) ) { }     // Non-compliant - always true
if ( ( s8a < 10 ) &&
     ( s8a > 20 ) ||                        // Non-compliant - left operand of ||
     ( s8b == 5 ) ) { }                     //                 always false

const uint8_t N = 4u;

if ( N == 4u )                              // Non-compliant - compiler is permitted
{                                           // to assume that N always has value 4
}

extern const volatile uint8_t M;

if ( M == 4u )                              // Compliant - compiler assumes M may
{                                           // change, even though the program
}                                           // cannot modify its value

while ( s8a > 10 )
{
  if ( s8a > 5 ) { }                        // Non-compliant - s8a always > 5

  --s8a;
}

for ( s8a = 0; s8a < 130; ++s8a ) { }       // Non-compliant - always true

while ( true ) { /* Do something */ }       // Compliant by exception #1

do { } while ( false );                     // Compliant by exception #2
                                            //   - if expanded from a macro

uint16_t n;                                 // Assume 10 <= n <= 100
uint16_t sum;

sum = 0;

for ( uint16_t i = ( n - 6u ); i < n; ++i )
{
  sum += i;
}
```


Licensed to: gunjanjain
22 Apr 2025

```
if ( ( sum % 2u ) == 0u )
{
  // Non-compliant - the sum of six, consecutive, non-negative integers is always
  // an odd number, so the controlling expression will always be false.
}

template< typename T >
void foo()
{
  if constexpr ( std::is_integral< T >() ) // Rule does not apply
  {
    // Handle integral case
  }
  else
  {
    // Handle other case
  }
}

template void foo< int >();
template void foo< float >();
```

See also

Rule 0.0.1

4.0.1 Unused values [misra]

Rule 0.1.1 A value should not be *unnecessarily written* to a local object

Category Advisory

Analysis Undecidable, System

Amplification

This rule applies to all accesses, either direct or through a pointer or reference, to objects with *automatic storage duration* that:

1. Have *trivially destructible* types (including basic types and enumeration types); or
2. Are arrays of *trivially destructible* types; or
3. Are STL containers (including `std::string`), where the *value_type* is *trivially destructible*.

The rule also applies to accesses to subobjects or elements of such objects.

An object is *unnecessarily written* when on each feasible path:

1. The object is destroyed before being *observed*; or
2. The object is written to again before being read.

An object is *observed* within an expression if its value affects the external state of the program, the control flow of the program, or the value of a different object.

Licensed to: gunjanjain
22 Apr 2025

The following examples illustrate different types of access to an object `i`:

```
int32_t f( int32_t j );

int32_t i = f( 1 );     // Written
i;                      // Read
i = 0;                  // Written (even if 'i' was 0 before the assignment)
auto j = i;             // Read and observed
++i;                    // Read and written
i += 3;                 // Read and written
i = i + j;              // Read and written
auto k1 = ++i;          // Read, written, read and observed
auto k2 = i++;          // Read, observed and written
arr[ i ] = f( 1 );      // Read and observed
if ( i ) { }            // Read and observed
( void )f( i );         // Read and observed
```

Observing any element of a container is considered to *observe* the full container and all of its elements. *Observing* a subobject is considered to *observe* the full object and all of its subobjects. Additionally, an object that is created outside of an *iteration statement* is considered to be *observed* (but not read) at the end of the *iteration statement*, provided it is also *observed* during any iteration.

A function's compliance with this rule is determined independently of the context in which the function is called. For example, a Boolean parameter is treated as if it may have a value of `true` or `false`, even if all the calls expressed in the current program use a value of `true` — see example `f4`, below.

Rationale

Giving an object a value that is never subsequently used is inefficient, and may indicate that a coding defect is present. Such writes are referred to as *dataflow anomalies*:

1. A *DU (Define–Use) dataflow anomaly* is present if a value that is written is *never observed*;
2. A *DD (Define–Define) dataflow anomaly* is present if a value overwrites another value before it has been read.

Within a loop, a value may be written to an object with the intent that it will be *observed* during the next iteration, meaning that the value written on the last iteration may never be *observed*. Whilst it is possible to restructure the loop to avoid this behaviour, there is a risk that the resulting code may be of lower quality (less clear, for example). This rule therefore considers *observation* during any iteration to apply to all values written to such an object, including a value written during the last iteration of a loop that is not actually *observed* — see example `f3`, below.

Observing part of a bigger object is considered to *observe* the object in its entirety; it is common to have code that operates on objects as a whole (initializing or writing to all subobjects), even if the value of only some of its subobjects are actually read. Requiring fine-grained writes would break encapsulation — see examples `f5` and `f6`, below.

A function, assuming its preconditions are respected, should always behave as specified. This is true irrespective of the calling context, including possible contexts that are not expressed in the current program. For this reason, path feasibility (within this rule) is determined without taking the actual calling contexts into consideration.

Exception

Even though the values passed as arguments to functions are written to their corresponding parameter objects, it is permitted for function parameters to remain *unobserved* when the function returns. This exception prevents crosstalk with Rule 0.2.2 which requires, in a decidable way, that function parameters are used. Note that writing to an unread parameter in a function body is a DD anomaly, which is a violation of this rule.

Licensed to: gunjanjain
22 Apr 2025

Example

```
int32_t f1( int32_t i )
{
  auto j = i;       // Non-compliant - j is not observed after being written

  i++;              // Non-compliant - i is not observed after being written

  return 0;
}

int32_t f2( int32_t i )
{
  auto & j = i;     // Rule does not apply to j, which is not an object

  j++;              // Compliant - writes object i that is observed in the return

  return i;
}

int32_t f3( int32_t j, int32_t k, int32_t m )
{
  for ( int32_t i = 0; i < 10; ++i )  // Compliant - i is observed in i < 10
  {
    m = 10;         // Non-compliant - when looping, overwrites incremented value
    ++k;            // Non-compliant - k is never observed

    use( j );       // Observation of j inside of the loop

    ++j;            // Compliant - observation above is sufficient for final write

    ++m;            // Compliant - observed in the return
  }                 // j is considered observed here as it was observed in the loop

  return m;
}

int32_t f4( bool    b,
            int32_t i,
            int32_t j )  // Compliant by exception - j is never observed
{
  i = 0;             // Non-compliant - value passed is overwritten

  int32_t k = 4;     // Compliant - value is observed in one feasible path

  if ( b )           // Both branches are considered feasible, even if the function
  {                  // is only called with b set to true
    return i;
  }
  else
  {
    return k;
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

```
struct Point { int32_t x; int32_t y; int32_t z; int32_t t; };

int32_t f5()
{
  Point p {};        // Compliant - p and its subobjects are observed in the return

  p.x = 2;
  p.x = 3;           // Non-compliant - overwrite the value 2 that is never read
  p.z = 4;           // Compliant - p.z is observed in the return

  return p.y;        // Observation of p.y also observes p, p.x, p.z and p.t
}

int32_t f6()
{
  std::vector< int32_t > v( 4, 0 );  // Compliant - v and its elements are observed
                                     //             in the return

  v[ 0 ] = 2;
  v[ 0 ] = 3;        // Non-compliant - overwrite the value 2 that is never read
  v[ 2 ] = 4;        // Compliant - v[ 2 ] is observed in the return

  return v[ 1 ];     // Observation of v[ 1 ] observes v and all of its elements
}

void f7( std::mutex & m )
{
  std::scoped_lock lock { m };  // Rule does not apply - destructor is non-trivial
}

char f8( bool b )
{
  char c = f( 1 );   // Non-compliant - assigned value never read

  if ( b )
  {
    c = 'h';         // The value of c is overwritten here

    return c;
  }
  else
  {
    return '\0';     // The value of c is not observed here
  }
}

void callee( int32_t & ri )
{
  ri++;              // Rule does not apply - reference is not an object
}

void caller()
{
  int32_t i = 0;

  callee( i );       // Non-compliant - i written and not subsequently observed
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 0.1.2 The value returned by a function shall be *used*

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule only applies when the function is called explicitly using *function call* syntax.

Rationale

It is possible to call a function without *using* the return value, which may be an error. If the return value of a function is intended to be explicitly discarded, it should be cast to `void` to ensure that it is *used*.

Overloaded operators are excluded from this requirement, as they should behave in the same way as built-in operators.

Note: this rule effectively requires all non-`void` functions to be treated as if they were declared `[[nodiscard]]`.

Example

```
uint16_t func();

void discarded()
{
  func();                          // Non-compliant - implicitly discarded
  ( void )func();                  // Compliant - void cast is a use
  auto b = func();                 // Compliant - used as initializer
}

void f1( std::string q )
{
  std::string s { q } ;            // Rule does not apply - not function call syntax
  s = q;                           // Rule does not apply - not function call syntax
  s.operator=( q );                // Non-compliant
}

void f2( std::function< int() > & f )
{
  f();                             // Non-compliant - using function call syntax

  auto a = []() { return 10; };
  a();                             // Non-compliant - using function call syntax
}
```

See also

Rule 28.6.4

4.0.2 Unused declarations [misra]

Rule 0.2.1 Variables with *limited visibility* should be *used* at least once

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

A variable has *limited visibility* if it is not a function parameter, and it has internal linkage or no linkage.

Licensed to: gunjanjain
22 Apr 2025

A variable is *used* when:

1. It is part of an *id-expression*; or
2. The variable is of `class` type and has a *user-provided* constructor or a *user-provided* destructor.

Rationale

Variables that are declared and never used within a project do not contribute to program output; they constitute noise and may indicate that the wrong variable name has been used or that one or more statements are missing.

Note: this rule allows the introduction of variables for the sole purpose of providing scoped resource allocation and release. For example:

```
{
  std::lock_guard< std::mutex > lock { mutex };    // Compliant - has user-provided
                                                   //             constructor
  // ...
} // User-provided destructor implicitly called here
```

Exception

This rule does not apply to:

1. Variables that have at least one *declaration* with the `[[maybe_unused]]` attribute.
2. Constant variables at namespace scope that are declared within a *header file*.

Example

```
class C { };                    // No user-provided constructor or destructor

namespace
{
  C c;                          // Non-compliant - unused
}

void maybeUnused( int32_t a )
{
  [[maybe_unused]]
    bool b = a > 0;             // Compliant (by exception #1 if NDEBUG is defined)

  assert( b );                  // Does not use b if NDEBUG is defined

  usefn( a );
}

const int16_t x = 19;           // Compliant - x is read in initializedButNotUsed
const int16_t y = 21;           // Non-compliant - would be compliant by exception #2
                                //                 if declared in a header file

void initializedButNotUsed()
{
  int16_t local_1 = 42;         // Non-compliant - local_1 is never read
  int16_t local_2;              // Compliant

  local_2 = x;                  // Use of local_2 for the purposes of this rule
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void userProvidedCtor()
{
  std::ifstream fs { "cfg.ini" };    // Compliant - user-provided constructor
}
```

See also

Rule 0.2.2

Rule 0.2.2 A named function parameter shall be *used* at least once

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to parameters that are declared `[[maybe_unused]]`.

Note: this rule also applies to the parameters of a lambda.

Rationale

It is expected that most functions will *use* their parameters. If a function parameter is *unused*, it is possible that the implementation of the function may not satisfy its requirements. This rule helps to highlight such potential mismatches.

In cases where an *unused* parameter is required, for example when defining a virtual function or a *callback* function, the parameter can be left unnamed. Where the use of a parameter depends on the expansion of a macro or varies between different template instantiations, then the parameter can be declared `[[maybe_unused]]`.

Example

```
class B
{
public:
  virtual int16_t f( int16_t a, int16_t b );
};

class D1 : public B
{
public:
  int16_t f( int16_t a, int16_t b ) override         // Non-compliant - 'b' unused
  {
    return a;
  }
};

class D2 : public B
{
public:
  int16_t f( int16_t a,                   // Compliant - 'a' is used
             int16_t ) override           // Rule does not apply - unnamed parameter
  {
    return a;
  }
};
```


Licensed to: gunjanjain
22 Apr 2025

```
class D3 : public B
{
public:
  int16_t f( int16_t a, int16_t b ) override         // Compliant
  {
    return a + b;
  }
};

class D4 : public B
{
public:
  int16_t f( int16_t a,                              // Compliant
             int16_t b [[maybe_unused]] ) override   // Rule does not apply -
                                                     //   declared [[maybe_unused]]
  {
    assert( b > 0 );                        // assert macro may expand to nothing,
                                            // leaving 'b' unused.
    return a;
  }
};

void f1(int32_t i,                          // Non-Compliant
        int32_t j )                         // Compliant - explicitly cast to void
{
  ( void )j;

  auto l = []( int32_t m,                   // Compliant
               int32_t n )                  // Non-compliant
  {
    return m;
  };
}

template< bool b >
int32_t f2( int32_t i,                      // Non-compliant for f2< false >
            int32_t j [[maybe_unused]] ) // Rule does not apply - [[maybe_unused]]
{
  if constexpr ( b )
  {
    return i + j;
  }

  return 0;
}
```

See also

Rule 0.2.1

Rule 0.2.3 Types with *limited visibility* should be *used* at least once

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

A type has *limited visibility* if it is declared in block scope or in unnamed namespace scope.

For the purposes of this rule:

- Type aliases, primary class templates, and alias templates are considered types.
- The closure type associated with a lambda is always *used*.

Licensed to: gunjanjain
22 Apr 2025

- A type is *used* if it is referenced within the *translation unit* outside of its definition.
- An enumeration type is *used* if any of its enumerators are *used*.
- An anonymous union is *used* if any of its members are *used*.
- The definition of a type includes the definition of its members and hidden friends.
- The definition of a class template includes its partial and explicit specializations.

Rationale

If a type is declared but not *used*, then it is unclear to a reviewer if the type is redundant or it has been left unused by mistake.

Exception

This rule does not apply to:

1. Types that have at least one *declaration* with the `[[maybe_unused]]` attribute.
2. Template parameters.
3. Partial or explicit specializations of class templates.

Example

```
int16_t f1()
{
  using T1 = int16_t;                    // Non-compliant
  using T2 [[maybe_unused]] = int32_t;   // Compliant by exception #1

  return 67;
}

namespace
{
  struct A1 { A1 f(); };                 // Compliant
  struct A2 { A2 f(); };                 // Non-compliant

  struct A2;                             // Not a use of A2

  A2 A2::f() { return *this; }           // Not a use of A2

  template< typename T >                 // Compliant by exception #2
  void foo()
  {
    A1 a;                                // Use of A1
    a.f();                               //  - even if foo is not instantiated
  }
}

template< bool cond >
inline auto foo()
{
  struct res { int32_t i; };             // Compliant

  if constexpr ( cond )
  {
    return 42;
  }
  else
  {
    return res { 42 };                   // res is utilized, even if cond is true
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

```
template< typename >
int32_t bar()
{
  return 42;
}

int32_t f2()
{
  return bar< struct P >();             // Compliant - P is used
}

namespace
{
  template< typename > struct C1 {};    // Non-compliant
                                        //  - C1 only utilized in its definition

  template<> struct C1< int32_t >       // Compliant by exception #3
  {
    void mbr()
    {
      C1< char > cc;
    }
  };
}

namespace
{
  template< typename > struct C2 {};    // Compliant - C2< float > used

  template<> struct C2< int32_t >;      // Compliant by exception #3

  C2< float > cf;                       // Use of C2
}

namespace
{
  static union                          // Non-compliant
  {
    int32_t i1;
    int32_t j1;
  };

  static union                          // Compliant
  {
    int32_t i2;
    int32_t j2;
  };
}

void f3()
{
  ++i2;                                 // Uses the anonymous union holding i2
}

namespace
{
  void f4()
  {
    []( auto ){};                       // Compliant - closure type is always used
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 0.2.4 Functions with *limited visibility* should be *used* at least once

Category Advisory

Analysis Decidable, System

Amplification

A function has *limited visibility* if it:

1. Is declared in an anonymous namespace; or
2. Is a member of a class in an anonymous namespace; or
3. Has namespace scope and is declared `static`; or
4. Is a `private`, non-`virtual` member.

A function is *used* when:

1. Its address is taken (including by reference); or
2. It is called; or
3. It is an operand of an expression in an unevaluated context; or
4. Another function in the same overload set is *used*.

This rule does not apply to:

1. *Special member functions*;
2. Functions defined as `= delete`.

Rationale

Functions with *limited visibility* are not generally used within an extensible API. If they are present but remain unused, then there may be an issue in the software design.

Unused functions in an overload set are acceptable as it allows the set to be internally consistent.

Exception

Functions that have at least one *declaration* with the `[[maybe_unused]]` attribute are permitted to be unused as the intent is explicit.

Example

```
struct Foo
{
  int32_t m1()                      // Public - rule does not apply
  {
    return -1;
  }

  static int32_t m2()               // Class scope - rule does not apply
  {
    return 42;
  }

  Foo()
  {
    m3();
  }
```


Licensed to: gunjanjain
22 Apr 2025

```
private:
  void m3() { }                    // Compliant - called
  void m4() { }                    // Non-compliant - not used
  void m5() { }                    // Compliant - used by a friend

  friend void ( *f4() )();

protected:
  void m6() { }                    // Protected - rule does not apply
};

static void f1() { }               // Non-compliant - not used

namespace
{
  void f2() { }                    // Non-compliant - not used
}

static void f3() { }               // Compliant - address taken in f4()

void ( *f4() )()                   // Rule does not apply - visibility not limited
{
  Foo bar;

  bar.m5();

  return &f3;
}

namespace A
{
  struct C1 {};
  static void swap( C1 &, C1 & );  // Compliant - overload set for call in f5
}

namespace B
{
  struct C2 {};
  static void swap( C2 &, C2 & );  // Non-compliant
}

namespace
{
  template< typename T >
  void swap( T &, T & );           // Compliant - overload set for call in f5
}

void f5( A::C1 c1, A::C1 c2 )      // Rule does not apply - visibility not limited
{
  swap( c1, c2 );
}
```

4.0.3 Runtime failures [misra]

Dir 0.3.1 Floating-point arithmetic should be used appropriately

[ISO/IEC/IEEE 60559]

Category Advisory

Amplification

A tool should highlight all suspicious uses of floating-point arithmetic.

Licensed to: gunjanjain
22 Apr 2025

Rationale

The safe use of floating-point arithmetic requires a high level of numerical analysis skills and in-depth knowledge of the compiler and target hardware.

The incorrect use of floating-point may lead to problems related to the presence of positive and negative zero, loss of precision, NaNs (quiet and signalling), infinities, overflow, underflow, catastrophic cancellation, etc.

Example

```
bool myIsNaN( double d )
{
  return d == std::numeric_limits< double >::quiet_NaN();  // Always returns false
}

void f()
{
  float f1 = 1E38f;
  float f2 = 10.0f;
  float f3 = 0.1f;
  float f4 = ( f1 * f2 ) * f3;
  float f5 = f1 * ( f2 * f3 );   // Values in f4 and f5 are significantly different
}
```

Dir 0.3.2 A function call shall not violate the function's preconditions

Category Required

Rationale

Violating a function's implicit or explicit preconditions may lead to it exhibiting unexpected results or having *undefined behaviour*.

Example

```
float f( float a )
{
  return fmodf( a, 0.0f );      // 'fmodf' requires a non zero value
}

// Precondition for 'b1' is that 'v' is not empty
int32_t b1( std::vector< int32_t > const & v )
{
  return v.front ();
}

int32_t b2()
{
  std::vector< int32_t > v;

  return b1( v );            // Violates the precondition of b1
}
```


Licensed to: gunjanjain
22 Apr 2025

4.4 General principles

4.4.1 Implementation compliance [intro.compliance]

Rule 4.1.1 A program shall conform to ISO/IEC 14882:2017 (C++17)

[defns.well.formed]
[intro.compliance]
[MISRA Guidelines] / Table 3
[IEC 61508-7] / Table C.1
[ISO 26262-6] / Table 1

Category Required

Analysis Undecidable, System

Amplification

A conforming program shall be *well-formed*, meaning that it shall be constructed according to the syntax rules, diagnosable semantic rules, and the *one-definition rule*, as specified for C++17 within ISO/IEC 14882:2017 [8]. In addition, the program shall use only those features of the C++ language and its library that are specified within the C++ Standard.

The use of language extensions is not permitted by this rule.

Note: a conforming implementation usually generates a diagnostic if a program is not *well-formed*, but be aware that:

- The C++ Standard does not require a diagnostic for all constructs that are not *well-formed*;
- A diagnostic need not necessarily be an error but could, for example, be a warning;
- The program may be translated and an executable generated, even if the program is not *well-formed*.

Rationale

Undesirable behaviours associated with language features that are extensions or which are specified outside of the C++ Standard have not been considered during the development of the guidelines within this document.

The behaviour of a program that is not *well-formed* is unpredictable.

It is recognized that it is sometimes necessary to use language extensions in embedded systems. If an extension is used (subject to a deviation), then appropriate steps shall be taken to guarantee predictable behaviour. This may be documented in a deviation permit to aid reuse, as explained in MISRA Compliance [1].

Example

```
#warning "declaring an interrupt handler"    // Non-compliant
__interrupt void handler();                  // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

Rule 4.1.2 Deprecated features should not be used

[depr]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

Deprecated features are those identified in Annex D of the C++ Standard, excluding those in [depr.c.headers].

Rationale

Features are deprecated by the C++ Standard when they are superseded by safer or better alternatives, or are considered to exhibit undesirable behaviour. Features deprecated by a particular version of the C++ Standard may be withdrawn in a later version.

For example:

- The `<codecvt>` header was deprecated in C++17; and
- The *noexcept-specifier* `throw` was deprecated in C++17 (and removed in C++20).

Note: use of the C versions of the C++ Standard Library headers ([depr.c.headers]) is not prohibited as these headers provide features that are equivalent to the ones in the C++ versions.

Example

```
#include <codecvt>          // Non-compliant - [depr.locale.stdcvt]


void foo() throw()          // Non-compliant - [depr.except.spec]
{
}
```

In the following example, the generation of the copy constructor of `C1` is deprecated when the destructor is user-declared:

```
struct C1
{
  ~C1() = default;
};

C1 c1a {};                  // Compliant - no use of copy constructor
C1 c1b { c1a };             // Non-compliant - [depr.impldec]
```

Rule 4.1.3 There shall be no occurrence of *undefined* or *critical unspecified behaviour*

[intro.abstract]

Category Required

Analysis Undecidable, System

Amplification

Many *undefined* and *unspecified behaviours* are covered by specific guidelines. This rule targets all other *undefined* and *critical unspecified behaviour*.

Licensed to: gunjanjain
22 Apr 2025

An *unspecified behaviour* within the C++ Standard is considered to be *critical* when it impacts the observable behaviour of the *abstract machine*.

For example:

- [expr.static.cast]/9 states that the value resulting from the explicit conversion of a scoped enumeration type to an integral type is *unspecified* if the value cannot be represented — this is *critical* as it will have an impact on observable behaviour.
- [global.functions]/1 states it is *unspecified* whether any non-member functions in the C++ Standard Library are defined as *inline* — this is not *critical* as it will not have an impact on observable behaviour.

Rationale

It is not possible to reason about the behaviour of any program that contains instances of *undefined behaviour*. In addition, any program that contains instances of *unspecified behaviour* is not guaranteed to behave predictably. These types of behaviour can be particularly difficult to detect during testing as the program may appear to behave as expected for a given set of test data.

Many of the guidelines within this document have been designed to help ensure that certain *undefined* and *unspecified behaviours* are avoided. However, other behaviours are not covered by specific guidelines — for example, because there is no practical guidance that can be given, other than the obvious statement that the behaviour should be avoided.

Note: an implementation is permitted to provide well-defined behaviour for a behaviour that is otherwise stated within the C++ Standard as being *undefined* or *unspecified*. It will be necessary to raise a deviation against this rule if any such well-defined behaviour is relied upon, including by means of the use of a language extension.

Example

The following examples are non-compliant:

```
u32a >> u32b                  // Undefined behaviour if u32b > 31
static_cast< int8_t >( 128 )  // Unspecified - 128 is not representable as int8_t
```

4.4.6 Program execution [intro.execution]

Rule 4.6.1 Operations on a memory location shall be sequenced appropriately

[intro.execution] Undefined 17

Category Required

Analysis Undecidable, System

Amplification

A side effect on a memory location shall not be *unsequenced* or *indeterminately sequenced* with respect to any other side effect on the same memory location, or any value computation using the value of any object in the same memory location.

For the purposes of this rule, all volatile accesses are considered to access a single, unique memory location.

Rationale

Unsequenced accesses to a memory location when one of the accesses has side effects results in *undefined behaviour*.

Licensed to: gunjanjain
22 Apr 2025

Additionally, *indeterminately sequenced* accesses could result in an expression yielding a different value for differing program states. This rule ensures that a program's behaviour is independent of the evaluation order (such as the evaluation of function arguments) chosen by the compiler.

An access to a volatile `v1` may have an effect on another, seemingly unrelated, volatile `v2`. For this reason, this rule considers all volatile accesses as if they were to a single, unique memory location.

Note: C++17 changed the evaluation order of several expressions from *indeterminately sequenced* to *sequenced before*, which means that code that is compliant with this rule may not work correctly with earlier versions of C++.

Example

```
char f( char & c, char a )
{
  c = a;

  return c;
}

void h( char a, char b );

char a;

h( f( a, 'a' ), f( a, 'b' ) );   // Non-compliant - value of a could be 'a' or 'b'
```

In the following example, `i` is read twice and modified twice. However, since C++17, the evaluation of the right-hand side of an assignment is *sequenced before* the evaluation of the left-hand side (see [expr.ass]), so all accesses to `i` occur in a defined order:

```
a[ i++ ] = b[ i++ ];             // Compliant in C++17
```

Even though there is no *undefined behaviour* in the following examples, they are non-compliant as the uses of `i` are *indeterminately sequenced* with respect to their increments:

```
x = b[ i ] + i++;                // Non-compliant
x = func( i++, i );              // Non-compliant
```

In the following example, all accesses to volatile variables are considered to have side effects on the same memory location:

```
extern volatile uint16_t v1;
extern volatile uint16_t v2;

uint16_t t = v1 + v2;            // Non-compliant - indeterminately sequenced

v1 = v1 & 0x80u;                 // Compliant
```


Licensed to: gunjanjain
22 Apr 2025

4.5 Lexical conventions

4.5.0 MISRA [misra]

Rule 5.0.1 *Trigraph-like sequences* should not be used

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

Trigraph-like sequences occur when the following character sequences appear in the source code:

```
??=  ??/  ??'  ??(  ??)  ??!  ??<  ??>  ??-
```

Rationale

Trigraphs were removed from the language in C++17. However, to prevent possible confusion, the sequences should not be used as it is unclear whether their replacement is expected.

Example

```
const char * msg = "(Date format is ??-??-??)";     // Non-compliant
const char * msg = "(Date format is ?\?-?\?-?\?)";  // Compliant
```

See also

Rule 4.1.2

4.5.7 Comments [lex.comment]

Rule 5.7.1 The character sequence `/*` shall not be used within a C-style comment

Category Required

Analysis Decidable, Single Translation Unit

Rationale

C++ does not support the nesting of C-style comments, even though some compilers support this as a non-portable language extension. A comment beginning with `/*` continues until the first `*/` is encountered.

Any `/*` sequence occurring inside a C-style comment is a violation of this rule.

Licensed to: gunjanjain
22 Apr 2025

Example

Consider the following code fragment:

```
/* Some comment, end comment marker accidentally omitted

Perform_Critical_Safety_Function( X );
/* <- this is non-compliant */
```

In reviewing the code containing the call to the function, the assumption is that it is executed. However, because the end comment marker is missing, the call to `Perform_Critical_Safety_Function` will **not** be executed.

Dir 5.7.2 Sections of code should not be "commented out"

Category Advisory

Amplification

This directive applies to the use of both `//` and `/* ... */` style comments.

For the purposes of this directive, the use of `#if 0` is also considered to be "commenting out".

Rationale

Comments should only be used to explain aspects of the source code; they should not be used to record the history of changes to the source code.

In addition, whilst the nesting of C-style comments is not supported by the C++ Standard, it is supported by some compilers. This means that the commenting out of any code that contains comments may behave differently with different compilers (see Rule 5.7.1).

This directive is generally undecidable, as it is not always possible for a tool to determine if a comment contains explanatory text, a code example or commented out code.

Note: it is acknowledged that it may be useful to quote statements or expressions as part of a larger comment in order to document and explain some aspect of the program (e.g. clarifying the use of a function, or explaining the algorithm being implemented). Such usage is not the intended target of this directive.

Example

The following compliant example documents an API with the use of pseudo-code. It is assumed that code wrapped within the ```` markup is recognized as documentation and is not commented out code.

```
// You should not call lock/unlock directly, but through RAII:
// ```
//   void f( Data & d, MyMutex & m )
//   {
//     std::scoped_lock lock { m };
//     d.doSomething();
//   }  // m is automatically unlocked
// ```

struct MyMutex
{
  void lock();
  void unlock();
};
```


Licensed to: gunjanjain
22 Apr 2025

The following non-compliant example uses a comment to record code history.

```
// Bug 42 - this call used to be:
//   calculate ( z , y + 1 );
calculate ( x , y - 1 );
```

```
enum E
{
#if 0     // Non-compliant
  E_0     // - this is considered to be commented out code
#else
  E_1
#endif
};
```

See also

Rule 5.7.1

Rule 5.7.3 Line-splicing shall not be used in // comments

[lex.phases] Undefined 1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule is applied in translation phase 2, after multibyte characters have been mapped to the basic source character set during translation phase 1 (see [lex.phases]).

Rationale

Line-splicing occurs when the \ character is immediately followed by a new-line character. If a source line containing a // comment ends with a \ character in the basic source character set, the next line becomes part of the comment. This may result in the unintentional removal of code.

Example

In the following non-compliant example, the physical line containing the `if` keyword is logically part of the previous line and is therefore part of a comment.

```
void f( bool b )
{
  uint16_t x = 0U;    // comment \
  if ( b )
  {
    ++x;              // This is always executed
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

4.5.10 Identifiers [lex.name]

Rule 5.10.1 User-defined identifiers shall have an appropriate form

[macro.names] Undefined 2
[lex.name] NDR 3
[lex.key]
[extern.types]
[usrlit.suffix]
[namespace.std] Undefined 1
[namespace.posix] Undefined 1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

When introducing an identifier, it shall be formed according to the following rules:

1. A *universal-character-name* used at the start of an identifier shall be:
   a. In the range `[a-z]`, `[A-Z]` or `_` or
   b. Within the character class *XID_Start*, as defined by the Unicode standard *UAX #44 [13]*.
2. A *universal-character-name* within an identifier shall be:
   a. One of the characters allowed at the start of an identifier; or
   b. Within the character class *XID_Continue*, as defined by the Unicode standard *UAX #44* [13].
3. All identifiers shall conform to *Normalization Form C*, as specified in *ISO/IEC 10646* [7].
4. An identifier shall not contain a double underscore `__`.
5. An identifier that is not used as a literal suffix shall not start with `_`.
6. A user-defined literal suffix shall start with a single `_` and shall not be preceded by a space.

A macro identifier shall additionally only be formed using characters in the ranges `[A-Z]`, `[0-9]` and `_`.

Other identifiers shall additionally:

1. Not be defined in namespace `std`, `posix`, or `stdN`, where `'N'` is any number; and
2. Not appear in the list `defined`, `final`, `override`, `clock_t`, `div_t`, `FILE`, `fpos_t`, `lconv`, `ldiv_t`, `mbstate_t`, `ptrdiff_t`, `sig_atomic_t`, `size_t`, `time_t`, `tm`, `va_list`, `wctrans_t`, `wctype_t` or `wint_t`.

Note: this rule does not apply to template specializations, as they do not introduce new identifiers — see [temp.expl.spec].

Rationale

This rule prohibits the introduction of an identifier with a reserved name, and restricts the characters permitted within identifiers to a subset of those that are currently permitted by the C++ Standard. This subset is aligned with Unicode recommendations that are expected to be adopted in a future revision of the C++ Standard.

For macro names, this rule further restricts the set of permitted characters for the following reasons:

1. It enforces commonly accepted coding style;
2. It helps distinguishing macros from other identifiers;

Licensed to: gunjanjain
22 Apr 2025

3. It prevents collision with the name of an attribute defined within the C++ Standard or with any name defined in the C++ Standard Library, preventing *undefined behaviour* (even when the corresponding *header file* is not explicitly included);
4. It prevents collision with keywords or alternative representations, preventing *undefined behaviour*.

The restrictions always prohibit the use of identifiers that are only prohibited by the C++ Standard within certain contexts (and for which no diagnostic is required in some cases). This rule broadens the context in which these identifiers are not acceptable in order to reduce the risk of confusion.

Example

```
int32_t i( = 2;                    // Non-compliant - character \ufd3e (even though
                                   // it may compile)

#define identity(a) a              // Non-compliant - shall be in uppercase

void f()
{
  auto _i = 0;                     // Non-compliant - using a leading _, even at
                                   // local scope, is prohibited
}

void operator ""_km( long double );     // Compliant - will be called for 1.0_km
void operator ""mil( long double );     // Non-compliant - user-defined literal
                                        // suffixes shall start with _

double operator ""_Bq ( long double );  // Compliant
double operator "" _Bq( long double );  // Non-compliant - _Bq is preceded by a
                                        // space, making it a reserved identifier

namespace std42
{
  inline namespace a
  {
    int i;                         // Non-compliant - defined within namespace stdN
  }
}

auto final = 42;                   // Non-compliant

#include <cstdio>                  // Compliant - even though it introduces FILE

namespace std
{
  template <> struct hash< A >     // Rule does not apply
  {
    size_t operator()( const A & x ) const;
  };
}
```


Licensed to: gunjanjain
22 Apr 2025

4.5.13 Literals [lex.literal]

Rule 5.13.1 Within character literals and non raw-string literals, \ shall only be used to form a defined escape sequence or universal character name

[lex.ccon] Implementation 7, 9
[lex.string] / 15

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The escape sequences defined within the C++ Standard are:

```
\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \", \<Octal Number>, \x<Hexadecimal Number>
```

The universal character names are:

```
\u hex-quad, \U hex-quad hex-quad
```

Rationale

The use of an undefined escape sequence results in *implementation-defined behaviour*.

Example

```
void fn()
{
  const char * a = "\k";          // Non-compliant
  const char * b = "\b\u00E9";    // Compliant
}
```

Rule 5.13.2 Octal escape sequences, hexadecimal escape sequences and universal character names shall be terminated

[lex.charset]
[lex.icon] Implementation 2, 3

Category Required

Analysis Decidable, Single Translation Unit

Amplification

An octal escape sequence, hexadecimal escape sequence or universal character name shall be terminated by either:

- The start of another escape sequence or universal character name; or
- The end of the character constant or the end of a string literal.

Rationale

There is potential for confusion if an octal escape sequence, hexadecimal escape sequence or universal character name is followed by other characters. For example, the string literal `"\x1f"` is a single-character, zero-terminated string, whereas `"\x1g"` includes the two characters `'\x1'` and `'g'`. The

Licensed to: gunjanjain
22 Apr 2025

potential for confusion is reduced if every octal escape sequence, hexadecimal escape sequence or universal character name in a character constant or string literal is terminated.

Example

```
const char * s1 = "\1234";       // Non-compliant - \123 is not terminated
```

In the following, the strings pointed to by `s2`, `s3` and `s4` are equivalent to `"Ag"`.

```
const char * s2 = "\x41g";       // Non-compliant
const char * s3 = "\x41" "g";    // Compliant - terminated by end of literal
const char * s4 = "\x41\x67";    // Compliant - terminated by another escape
```

In the following, `s5` contains a universal character name consisting of four hex digits (`\u`), whilst `s6` contains a universal character name consisting of eight hex digits (`\U`).

```
const char * s5 = "\u0001F600";  // Non-compliant - \u0001 is not terminated
const char * s6 = "\U0001F600";  // Compliant - terminated by end of literal
```

Rule 5.13.3 Octal constants shall not be used

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Any integer constant beginning with a `0` (zero) is an octal constant. Because of this, a zero-prefixed constant that is intended to be a decimal number may be interpreted as an octal number, contrary to developer expectations.

Note: this rule does not apply to octal escape sequences because the use of a leading `\` character means that there is less scope for confusion.

Exception

The integer constant `0` (written as a single numeric digit) is an octal constant, but its use is permitted as an exception to this rule.

Example

```
code[ 1 ] = 109;      // Compliant     - decimal 109
code[ 2 ] = 100;      // Compliant     - decimal 100
code[ 3 ] = 052;      // Non-compliant - equivalent to decimal 42, not 52
code[ 4 ] = 071;      // Non-compliant - equivalent to decimal 57, not 71
code[ 5 ] = 0;        // Compliant by exception
code[ 6 ] = 000;      // Non-compliant - exception does not apply
code[ 7 ] = '\123';   // Rule does not apply
```

Rule 5.13.4 Unsigned *integer literals* shall be appropriately suffixed

[lex.icon]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to any *integer-literal* that exists after preprocessing. It does not apply to *user-defined-integer-literals*.

Licensed to: gunjanjain
22 Apr 2025

An unsigned *integer-suffix* is required when the type of the *integer literal*, as specified by the C++ Standard in [lex.icon], is unsigned.

Note: this rule does not depend on the context in which a literal is used; promotion and other conversions that may be applied to the value are not relevant in determining compliance with this rule.

Rationale

The type of an *integer literal* is a potential source of confusion, because it is dependent on a complex combination of factors including:

- The magnitude of the constant;
- The implemented sizes of the integer types;
- The presence of any suffixes;
- The number base that is used.

For example, the decimal *integer literal* `32768` always has signed type. However, the *integer literal* `0x8000` is of type `unsigned int` in a 16-bit environment, but of type `signed int` in a 32-bit environment. Adding a `U` or `u` suffix to the *integer literal* makes the signedness of the value explicit on a 16-bit platform.

Note: compliance checks against this rule will only be valid if an analysis tool has been configured with the same integer sizes as the compiler that is being used within the project.

Example

The following examples assume that `int` is 16 bits:

```
    auto x = 32768;     // Compliant - signed type
    auto y = 0x8000;    // Non-compliant - unsigned type
uint16_t z = 123;       // Compliant - 'u' is not required as '123' is signed

void f( uint16_t );     // #1
void f(  int16_t );     // #2

void b()
{
  f( 0x8000  );         // Non-compliant - calls #1 as 0x8000 is unsigned
  f( 0x8000u );         // Compliant - calls #1
  f( 0x7FFF  );         // Compliant - calls #2 as 0x7FFF is signed
  f( 0x7FFFu );         // Compliant - calls #1
}
```

Rule 5.13.5 The lowercase form of `L` shall not be used as the first character in a literal suffix

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to *user-defined-literal*s.

Rationale

Using the uppercase suffix `L` removes the potential ambiguity between `1` (digit 1) and `l` (lowercase `L`) when declaring numeric literals. The ambiguity only occurs when lowercase `L` is used as the first letter of a suffix.

Licensed to: gunjanjain
22 Apr 2025

Example

```
int64_t const    a = 0L;       // Compliant
int64_t const    b = 0l;       // Non-compliant

uint64_t const   c = 1Lu;      // Compliant
uint64_t const   d = 1lU;      // Non-compliant

uint64_t const   e = 2uLL;     // Compliant
uint64_t const   f = 2Ull;     // Compliant

long long const  g = 3LL;      // Compliant
long long const  h = 3ll;      // Non-compliant

long double const i = 1.2L;    // Compliant
long double const j = 3.4l;    // Non-compliant

constexpr Litre operator"" _l( long double val ) noexcept
{
  return Litre { val };        // Assumes type Litre is defined
}

auto volume = 42.1_l;          // Rule does not apply
```

Rule 5.13.6 An *integer-literal* of type `long long` shall not use a single `L` or `l` in any suffix

[lex.icon]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to both `signed long long` and `unsigned long long` literals.

Note: this rule does not apply to *user-defined-literals*.

Rationale

A literal with a suffix that has a single `L` could be a `signed` or `unsigned long long`. Use of the `LL` suffix for `long long` literals is more explicit and less error-prone.

Example

All of the following examples assume that `long` is 32-bits and `long long` is 64-bits.

```
auto k1 = 12345678998L;        // Non-compliant
auto k2 = 12345678998UL;       // Non-compliant
auto k3 = 12345678998ull;      // Compliant
auto k4 = 0xfeeddeadbeefL;     // Non-compliant
auto k5 = 0xfeeddeadbeefLL;    // Compliant
```

The rule does not apply to the following as the value is not `long long`:

```
auto k6 = 12345L;
auto k7 = 12345UL;
auto k8 = 0x0badc0deL;
```


Licensed to: gunjanjain
22 Apr 2025

The rule does not apply to the following as they do not have `L` or `l` suffixes:

```
auto k9 = 12345678998;
auto kA = 12345678998U;
```

See also

Rule 5.13.5

Rule 5.13.7 String literals with different encoding prefixes shall not be concatenated

[lex.string] Implementation 13

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The encoding prefixes are:

- `L` — wide string literal;
- `u8` — UTF-8 string literal;
- `u` — char16_t string literal;
- `U` — char32_t string literal.

For the purposes of this rule, an empty encoding-prefix is considered to be different to a non-empty encoding-prefix, even when they have the same meaning.

Note: the `R` prefix is not an encoding-prefix.

Rationale

Concatenation of string literals with different encoding prefixes is either *ill-formed* or *conditionally-supported* with *implementation-defined behaviour*. The behaviour related to the concatenation of string literals with and without encoding prefixes has changed as the C++ Standard has evolved. Concatenations of these forms are not permitted to ensure that the behaviour is as expected, especially in the presence of legacy code.

When concatenating a string literal with a prefix with one having no prefix, the behaviour is as if both have the same encoding prefix. For example, the concatenation `u8"" "\u00fc"` is equivalent to `u8"\u00fc"` (`0xc3 0xbc` — for some character set) and not `"\u00fc"` (`0xfc`), which may not meet developer expectations. This rule is therefore stricter than the C++ Standard, and considers an empty encoding-prefix to be different to a non-empty encoding-prefix.

Note: concatenation of string literals with different encoding prefixes is likely to become *ill-formed* in a future version of the C++ Standard.

Example

```
const char    * s0 =  "Hello"  "World";          // Compliant
const wchar_t * s1 = L"Hello" L"World";          // Compliant

const wchar_t * s2 =   "Hello" L"World";         // Non-compliant
const wchar_t * s3 =  u"Hello" L"World";         // Non-compliant - may not compile
                  // u8"Hello" L"World";         // Ill-formed

const char    * s4 = u8R"#(Hello)#" u8"World";   // Compliant
const char    * s5 = u8R"#(Hello)#"   "World";   // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

4.6 Basic concepts

4.6.0 MISRA [misra]

Rule 6.0.1 Block scope *declarations* shall not be *visually ambiguous*

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A block scope *declaration* is *visually ambiguous* when:

- It declares a function; or
- It declares an object with redundant parentheses surrounding the object's name.

Note: this rule does not apply to *Lambda expressions* as they are not function declarations.

Rationale

Due to the syntactic similarity of function declarations and object definitions that use parentheses for initialization, it is possible that a declaration may be misinterpreted by the developer. For example, a function declaration may be interpreted as an object definition, which is sometimes referred to as *the most-vexing parse*.

The C++ grammar allows for redundant parentheses around a declarator, where what appears to be the construction of an object with a single argument to the constructor is actually the declaration of an object of that "argument" name and a call to the default constructor.

Note: using braces instead of parentheses for object initialization, where possible, avoids the *most-vexing parse*.

Example

```
class A {};

void f1()
{
  void f2();        // Non-compliant - function declaration at block scope
  A a1();           // Non-compliant - appears to declare an object with no
                    // arguments to constructor, but it declares a function
                    // 'a1' returning type 'A' and taking no parameters.
  A a2;             // Compliant
}

int32_t j;

void f3()
{
  int32_t ( j );    // Non-compliant - declares 'j' (using redundant parentheses)
  int32_t { j };    // Compliant with this rule, but violates "See also"
}                   //   - Creates a temporary object with value 'j'.
```

See also

Rule 9.2.1

Licensed to: gunjanjain
22 Apr 2025

Rule 6.0.2 When an array with external linkage is declared, its size should be explicitly specified

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to non-defining *declarations* only. It is possible to define an array and specify its size implicitly by means of initialization.

Rationale

Although it is possible to declare an array with incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined. Providing size information for each *declaration* permits them to be checked for consistency. It may also permit a static checker to perform some array bounds analysis without needing to analyse more than one *translation unit*.

Example

```
       int32_t array1[ 10 ];                // Compliant
extern int32_t array2[ ];                   // Non-compliant
       int32_t array3[ ] = { 0, 10, 15 };   // Compliant
extern int32_t array4[ 42 ];                // Compliant
```

Rule 6.0.3 The only *declarations* in the global namespace should be `main`, namespace declarations and `extern "C"` declarations

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule also prohibits use of *using directives* and *inline namespaces* in the *global namespace*.

It does not apply to *namespace aliases*, `static_assert` or to names that are declared within the C++ Standard.

Rationale

Declaring names into appropriate *namespaces* reduces the names found during lookup, helping to ensure that the names found meet developer expectations.

Adherence with this rule is particularly important within *header files*, as it reduces the chance that the order of their inclusion will affect program behaviour.

Notes:

1. *Using directives* and *inline namespaces* do not actually add names to the global namespace, but they do make them appear as if they are in it.
2. Names declared within an *anonymous namespace* appear in the *global namespace*. However, their use is permitted as they do not have external linkage.

Example

```
void f1( int32_t );        // Non-compliant

int32_t x1;                // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

```
namespace                   // Compliant
{
  void f2( int32_t );       // Rule does not apply

  int32_t x2;               // Rule does not apply
}

namespace MY_API            // Compliant
{
  void b2( int32_t );       // Rule does not apply

  int32_t x2;               // Rule does not apply
}

using namespace MY_API;     // Non-compliant
using MY_API::b2;           // Non-compliant
namespace MY = MY_API;      // Compliant

int main()                  // Compliant
{
  extern void f3();         // Non-compliant
}
```

Rule 6.0.4 The identifier *main* shall not be used for a function other than the global function `main`

[basic.start.main] Implementation 2, 3

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule also applies to any other entry points defined by the implementation.

Rationale

`main` (or its equivalent) is the entry point to the program and is the only identifier which must be in the global namespace. The use of *main* for other functions may not meet developer expectations.

Example

```
int main()        // Compliant
{
}

namespace
{
  int main()      // Non-compliant
  {
  }
}

namespace NS
{
  int main()      // Non-compliant
  {
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

4.6.2 One-definition rule [basic.def.odr]

In essence, the requirement for the *one-definition rule* arises because C++ compilers usually treat each source file (with any included *header files*) as separate *translation units*, where each *translation unit* is compiled in isolation. The set of compiled *translation units* is then linked together to form the executable program.

The linker is allowed to assume that objects, templates, types, etc. that share the same name in different *translation units* refer to the same definition. There are no guarantees that violations of that assumption will be diagnosed.

In the following example, the same `struct S` appears to be defined in both *translation units*, but the definitions are not the same, so the result is not what the developer expects.

```
// Program A: file1.cpp
struct S
{
  int32_t x;
  int32_t y;
};

int32_t XminusY( S & s )
{
  return ( s.x - s.y );
}

// Program A: file2.cpp
struct S
{
  int32_t y;  // Note that the order of x and y is different
  int32_t x;
};

void setX( S & s, int32_t v ) { s.x = v; }
void setY( S & s, int32_t v ) { s.y = v; }
```

This program contains *undefined behaviour*; one possible outcome is that the result of `XminusY` is `y - x`.

As stated above, the linker is not required to check the compatibility of the two definitions; the *one-definition rule* puts the onus on the developer to ensure that the definitions are compatible. The rules within this section reinforce the need to follow the *one-definition rule*, and provide specific instructions to the developer.

Rule 6.2.1 The *one-definition rule* shall not be violated

[basic.def.odr] Undefined 6.6

Category Required

Analysis Decidable, System

Rationale

Violation of the *one-definition rule* (*ODR*) results in *undefined behaviour* — for example, when an *entity* has:

- No definition; or
- Multiple non-inline definitions in different *translation units*; or
- Multiple inline definitions in different *translation units* that are not the same; or
- Different initializer values.

Licensed to: gunjanjain
22 Apr 2025

Example

```
// File a.cpp
struct S1
{
  int32_t i;
};

struct S2
{
  int32_t i;
};

// File b.cpp
struct S1
{
  int64_t i;
};                          // Non-compliant - definitions of S1 are not the same

struct S2
{
  int32_t i;
  int32_t j;
};                          // Non-compliant - definitions of S2 are not the same
```

The following example is non-compliant as `File1.cpp` and `File2.cpp` introduce different definitions of `h`, with the call they contain to `f` resolving to different overloads in each definition.

```
// File1.h
void f( int32_t i );

// File2.h
void f( int64_t i );

// File3.h
inline void h( int64_t i )
{
  f( i );                   // Nested call
}

// File1.cpp
#include "File1.h"
#include "File3.h"

void f1()
{
  h( 42 );                  // Nested call in h is to int32_t overload of f
}

// File2.cpp
#include "File2.h"
#include "File3.h"

void f2()
{
  h( 42 );                  // Nested call in h is to int64_t overload of f
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 6.2.2 All *declarations* of a variable or function shall have the same type

[basic.def.odr] NDR 2; Undefined 5
[dcl.link]
[over.load] / 2.1
[dcl.attr.noreturn] NDR 1

Category Required

Analysis Decidable, System

Amplification

Two variable *declarations* with the same name refer to the same variable if they have the same scope. Two function *declarations* with the same name refer to the same function if they have the same scope and have equivalent parameter declarations (see [over.dcl]/1). *Declarations* of variables in the global scope and *declarations* of variables and functions with C linkage that have the same identifier declare a single *entity* (note there is no overloading in C).

For the purposes of this rule:

1. An array declared with an unknown bound has the same type as an array declared with the same element type and a known bound; and
2. A pointer to an incomplete type has the same type as a pointer to the complete type.

The following restrictions apply:

1. When several *declarations* of the same *entity* exist, they shall have the same type;
2. All *declarations* of a function declared with the `[[noreturn]]` attribute shall have that attribute (see [dcl.attr.noreturn]).

Note: functions with C linkage are always distinct from functions with C++ linkage.

Rationale

It is *undefined behaviour* if the *declarations* of a variable or function in two different *translation units* do not have the same type.

While attributes are not part of a function type, inconsistent use of the `[[noreturn]]` attribute results in an *ill-formed (no diagnostic required)* program.

Example

All the *declarations* of `f3` in the following files conflict with each other and are non-compliant.

```
// File a.cpp
typedef int32_t myint;
extern     int32_t a;                // Non-compliant - see b.cpp
extern     int32_t b [];             // Compliant
extern     char    c;                // Non-compliant - see b.cpp
extern     int32_t d;                // Compliant
extern     myint e;                  // Compliant

           int32_t f1();             // Non-compliant - see b.cpp
           int32_t f2( int32_t );    // Compliant
extern "C" int32_t f3( int32_t );    // Non-compliant
           int32_t f4();             // Non-compliant - see b.cpp
```


Licensed to: gunjanjain
22 Apr 2025

```
// File b.cpp
extern     int64_t a;                // Non-compliant - see a.cpp
extern     int32_t b [ 5 ];          // Compliant
           int16_t c;                // Non-compliant - see a.cpp
           int32_t d { 1 };          // Compliant
           int32_t e;                // Compliant

           char f1();                // Non-compliant - see a.cpp
           char f2( char );          // Compliant - not the same function as
                                     //             int32_t f2( int32_t )
extern "C" int32_t f3( char );       // Non-compliant
           int32_t f4() noexcept;    // Non-compliant - see a.cpp
                                     //   Different exception specification

// File c.cpp
extern "C" int32_t f3;               // Non-compliant

// File d.cpp
int32_t f3;                          // Non-compliant
```

Rule 6.2.3 The source code used to implement an *entity* shall appear only once

[basic.def.odr] Undefined 6.6; NDR 4

Category Required

Analysis Decidable, System

Amplification

For the purposes of this rule, an *entity* is a variable, type, function, or template thereof.

Note: multiple different *specializations* for the same *primary template* and multiple overloads for a function with the same name but with different signatures are different *entities*.

This rule requires that the source code used to implement an *entity* shall appear only once within a project. If the entity is *inline*, it can be implemented within a *header file*; it is permitted to include such a *header file* in multiple *translation units*.

Additionally, *explicit specializations* of templates shall either be implemented in the same file as the *primary template*, or in a file where one of the fully specialized arguments is defined.

Note: an *entity* may have no implementation — for example, an incomplete type does not need a definition when it is used as a tag.

Rationale

Non-inline *entities* shall only be defined once in a program. Inline *entities* can be defined once for each *translation unit*, but the definitions shall be identical. This principle is known as the *one-definition rule*.

Requiring that the source code for the definition of any *entity* appears only once reduces the risk of violating the *one-definition rule* and makes the code simpler.

The declaration of a template's *explicit specialization* must be visible when it matches the arguments of the template that is being instantiated, otherwise, an *implicit specialization* will be generated, violating the *one-definition rule*. Implementing an *explicit specialization* in the same file as the *primary template* or the argument for which it is specialized ensures that this constraint is satisfied.

Licensed to: gunjanjain
22 Apr 2025

Example

```
// file1.h
inline int16_t i = 10;

// file2.h
inline int16_t i = 10;              // Non-compliant - two definitions of i
```

The following example demonstrates inconsistent definitions of b:

```
// file1.cpp
int16_t b;                          // Non-compliant - ill-formed (see file2.cpp)

// file2.cpp
int32_t b;                          // Non-compliant - ill-formed (see file1.cpp)
```

In the following example, the full template specialization within a different file results in a violation of the *one-definition rule* (which is not the case for the template specialization A< D >, as that is within the file that defines D):

```
// a.h - #include guard omitted for brevity
template< typename T >
class A {};

// b.h
#include "a.h"

A< int32_t > const a1 {};

// c.h
#include "a.h"

template<>
class A< int32_t > {};              // Non-compliant

// d.h
#include "a.h"

class D {};

template<>
class A< D > {};                    // Compliant

// main.cpp
#include "b.h"
#include "c.h"                      // ODR violation
#include "d.h"
A< D > const a2 {};                 // OK - requires inclusion of d.h
```

In the following example, the partial template specialization within a different file results in a violation of the *one-definition rule*:

```
// wrap.h
template< typename V >
struct wrap
{
  V value;
};

// wrap_ptr.h
#include "wrap.h"

template< typename V >
struct wrap< V * > {}               // Non-compliant - should be in wrap.h
```


Licensed to: gunjanjain
22 Apr 2025

```
// w.cpp
#include "wrap.h"                    // No specialization visible

wrap< char * > a_wrap;               // ODR violation - see wp.cpp

// wp.cpp
#include "wrap_ptr.h"                // Specialization visible

wrap< char * > b_wrap;               // ODR violation - see w.cpp
```

Rule 6.2.4 A *header file* shall not contain definitions of functions or objects that are non-inline and have external linkage

[basic.def.odr]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule prohibits the definition within a *header file* of non-inline:

- Namespace-scope variables; and
- Namespace-scope functions; and
- Static and non-static member functions; and
- Non-`const`, static data members.

This rule does not apply to *entities* without linkage (scope local *entities*) or *entities* with internal linkage.

Rationale

Header files should be used to declare C++ templates, types, functions, references and objects.

Defining a non-inline *entity* (function or object) with external linkage in a *header file* causes a violation of the *one-definition rule* when that *header file* is included in multiple *translation units*, resulting in *undefined behaviour*.

Whilst defining non-inline *entities* with internal linkage in *header files* can cause needless duplication, it is not a violation of this rule.

Entities defined explicitly or implicitly as *inline*, or without external linkage, can appear in *header files* without risking violation of the *one-definition rule* if all definitions across all *translation units* are consistent. The latter can be guaranteed by using a single *header file* to define such an *entity* (see Rule 6.2.3).

Example

```
// Header file a.h
       void f1();            // Rule does not apply - not a definition
       void f2() { }         // Non-compliant
inline void f3() { }         // Compliant

template< typename T >
void f4( T ) { }             // Compliant - implicitly inline

int32_t a;                   // Non-compliant

constexpr auto ans { 42 };   // Compliant - no external linkage
```


Licensed to: gunjanjain
22 Apr 2025

```
struct X
{
  int32_t a;                // Compliant - no linkage
  inline static const
  int32_t b { 2 };          // Compliant - X::b has external linkage but is inline
};
```

See also

Rule 6.2.1, Rule 6.2.3

4.6.4 Name lookup [basic.lookup]

Rule 6.4.1 A variable declared in an *inner scope* shall not hide a variable declared in an *outer scope*

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A variable *declaration* in an *inner scope* is considered to *hide* a variable in an *outer scope* when it has the same name and the variable in the *outer scope* would be found by name lookup in the *inner scope* at a point immediately before the *declaration*.

The terms *outer scope* and *inner scope* are defined as follows:

1. The global scope is the outermost scope;
2. Each block (*compound-statement*), namespace or class introduces an *inner scope*;
3. In a function definition, the function parameters have the same scope as the corresponding function body (*compound-statement* or *function-try-block*);
4. A derived class is treated as an *inner scope* with respect to the base class;
5. The definition of a member function introduces an *inner scope* to the class's definition;
6. The *selection-statements* and *iteration-statements* introduce an *inner scope* which contains the controlled statement(s) and corresponding *condition* and *init-statement*.

If *declarations* from a namespace are introduced into a scope by a *using-declaration*, then they are treated as though they were declared in that scope.

For the purposes of this rule, the following are treated as the *declaration* of variables:

1. All data member and function parameter *declarations*; and
2. The enumerators of an *unscoped enumeration type* (which have the same scope as the enumeration type).

Rationale

Identifier *hiding* may lead to developer confusion.

Note: this rule prevents the name of a global variable from being reused as the name of a local variable.

Exception

A class constructor may have a parameter with the same name as a member variable, provided the only use made of that parameter is to initialize the member. This is a common idiom that poses no risk.

Licensed to: gunjanjain
22 Apr 2025

Example

```
int16_t i;

void f1()
{
  int32_t i;                // Non-compliant - hides i in global scope
  int32_t z;

  if ( i == 3 )             // It could be confusing as to which i this refers
  {
    int32_t z;              // Non-compliant - hides z before if
  }
}

void f2( int8_t i )         // Non-compliant - hides i in global scope
{
}

class C
{
  float i;                  // Non-compliant - hides i in global scope
  float j;

public:
  C ( float j )             // Compliant by exception
    : j ( j ) {}

  C ( float j, float k )
    : j ( j )
  {
    j += k;                 // Non-compliant - 'j' hides C::j
  }

  void f3()
  {
    int32_t j = 0;          // Non-compliant - hides C::j
  }
};

namespace NS1
{
  int32_t i;                // Non-compliant - hides i in global scope

  void f4( int32_t j )      // Compliant - parameter j does not hide C::j
  {
    int32_t l = i + j;      // Compiles using ::i if NS1::i declaration removed
  }
}

namespace NS2
{
  int32_t v;
}

using NS2::v;

void f5()
{
  float v;                  // Non-compliant - using hides NS2::v in global scope
}
```


Licensed to: gunjanjain
22 Apr 2025

```
enum E { e0, e1, e2 };

namespace
{
  int32_t e1 = 32;          // Non-compliant - hides e1 member of E (in global
}                           // scope)
```

Note that compiler reporting of a *redeclaration* error against `para` is inconsistent for the following example:

```
int16_t f6( int16_t para )  // 'para' has same scope as function body
try
{                           // Inner scope within function body
  int16_t para = 1;         // Non-compliant - hides parameter
  int16_t a    = 2;

  return para + a;
}

catch( ... )
{                           // Inner scope within function body
  int16_t para = 1;         // Non-compliant - hides parameter
  int16_t a    = 2;

  return para + a;
}

void f7( int32_t i )
{
  for ( int32_t i = 0; i < 9; ++i ) {}   // Non-compliant

  for ( int32_t j = 0; j < i; ++j ) {}

  for ( int32_t j = 0; j < i; ++j ) {}   // Compliant - new scope

  for ( int32_t k = 0; k < i; ++k ) {}

  int32_t k = i;                         // Compliant - for-loop 'k' not in scope

  for ( int32_t k = 0; k < i; ++k ) {}   // Non-compliant - hides 'k' above

  if ( get() )                           // Introduces an inner scope into which 'k'
  {                                      //   is defined.
    int32_t k;                           // Non-compliant - hides 'k' in outer scope
  }
}
```

In the following example, there is no hiding in the compliant examples as the local variable `z` cannot be found by name lookup within the body of a lambda.

```
void f8()
{
  char z;

  auto L1 = [ z ](){ return z; };         // Compliant - no hiding
  auto L2 = [](  char z ){ return z; };   // Compliant - no hiding
  auto L3 = [](){ char z { 'a' }; };      // Compliant - no hiding
  auto L4 = [ z ](){ char z { 'a' }; };   // Non-compliant - captured z is hidden
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 6.4.2 Derived classes shall not *conceal* functions that are inherited from their bases

[class.member.lookup]
[namespace.udecl] / 4, 15

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A function from a base class is *concealed* in the derived class if the derived class contains any function or variable with the same name, unless:

- The base class is inherited privately; or
- The base class function is virtual and the derived class contains an override of it; or
- The base class function is *introduced* into the derived class through a *using-declaration*; or
- The base class function is a *copy assignment operator* or a *move assignment operator*.

Note: this rule does not apply to constructors or destructors as they do not have names.

Rationale

When performing name lookup, if a function with the requested name exists in the derived class, no lookup will be performed in any base class, even if the base classes contain functions that would have been better matches. This may result in a call being made to an unexpected function.

Additionally, calling a function directly or through a base class pointer should result in the same function being called, which may not be the case when a non-virtual base class function is *concealed*.

Members of a class inherited privately are not accessible outside of the derived class, and so users of the derived type will not encounter the issues identified above.

Note: a *using-declaration* will only introduce an overload into a derived class if the derived class does not contain the same overload — see example for `f5`.

Example

```
class Base
{
public:
          void   f1( int32_t i );
          void   f2( int32_t i );
  virtual Base * f3( char    c );
          void   f4( char    c );
          void   f5( int32_t i );
          void   f5( char    c );
};

class Derived: public Base
{
public:
  // Compliant - does not conceal Base::operator=
  Derived & operator=( Derived const & ) &;

  // Non-compliant - Derived::f1 conceals Base::f1
  void f1( float f );

                          // Compliant - Base::f2 is not concealed
  using Base::f2;         // Introduces Base::f2( int32_t ) overload
  void f2( float f );     // Using declaration means this overload does not conceal
```


Licensed to: gunjanjain
22 Apr 2025

```
                                                // Compliant - Base::f3 is not concealed
  Derived * f3( char const c ) override;        // overrides Base::f3( char )
     void    f3( int32_t    i );

                                                // Non-compliant - Base::f4 is concealed
  void f4( char    c );                         // Not an override
  void f4( int32_t i );

                          // Non-compliant - Base::f5( int32_t ) is concealed
  using Base::f5;         // Introduces Base::f5( char ), not Base::f5( int32_t ) as
  void f5( int32_t i );   // this function has the same signature
};

class PrivateDerived: private Base
{
public:
  void f1( float f );     // Compliant - Base inherited privately
};
```

Rule 6.4.3 A name that is present in a dependent base shall not be resolved by unqualified lookup

[basic.lookup.unqual]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to names that would be found by unqualified lookup if none of the base classes were dependent.

Note: this rule does not apply to names from a base class that are introduced into the derived class through a *using* declaration.

Rationale

For a template class with a dependent base, the use of an unqualified name that does not refer to an *entity* in that class is taken to mean an *entity* in global scope, even if there is an *entity* with that name in the base. This differs from the behaviour in non-template classes, where the *entity* in the base will be selected in preference to an *entity* in global scope. This may not be consistent with developer expectations.

Note: using a *qualified-id* or prefixing the identifier with `this->` ensures that an *entity* in the base is selected.

Example

```
using int32_t = int;
using int16_t = short;

typedef int32_t Type;
void g();

template< typename T > struct B;

template< typename T >
struct A : B< T >
{
  using typename B< T >::ConstType;
```


Licensed to: gunjanjain
22 Apr 2025

```
  void f1()
  {
    // Non-compliant for A< int32_t > - compiler will choose ::Type
    //   If B were non-dependent, B< int32_t>::Type would have been chosen
    // Non-compliant for A< int16_t > - compiler will choose ::Type
    //   If B were non-dependent, B< int16_t>::Type would have been chosen
    Type t = 0;

    // Compliant - compiler finds the name introduced by the using declaration
    ConstType t = 0;

    // Non-compliant for A< int32_t > - compiler will choose ::g
    //   If B were non-dependent, B< int32_t>::g would have been chosen
    // Compliant for A< int16_t > - base B< int16_t > has no member g
    g();
  }

  void f2()
  {
    ::Type t1 = 0;                  // Compliant - explicit use of global Type
    ::g();                          // Compliant - explicit use of global g

    typename B< T >::Type t2 = 0;   // Compliant - explicit use of base Type

    // Compliant for A< int32_t > - uses base g
    // Compile error for A< int16_t >
    this->g();
  }
};

template< typename T >
struct B
{
  typedef T Type;
  typedef T const ConstType;
  void g();
};

template<> struct B< int16_t >
{
  typedef int16_t Type;
  typedef int16_t const ConstType;
};

template struct A< int32_t >;
template struct A< int16_t >;

using value_type = char16_t;

template< typename String >
class MyString : public String
{
  // Non-compliant for MyString<std::string> - compiler will choose ::value_type
  //   If MyString inherited directly from std::string, std::string::value_type
  //   would have been chosen
  value_type separator;
};

MyString< std::string > ms;
```


Licensed to: gunjanjain
22 Apr 2025

4.6.5 Program and linkage [basic.link]

Rule 6.5.1 A function or object with external linkage should be *introduced* in a *header file*

[basic.scope.namespace]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to functions and objects with *namespace scope*. The *header file* containing the *introduction* should be included by every *translation unit* in which it is defined or used.

This rule does not apply to the function `main`.

Rationale

Placing the *introductions* of functions and objects with external linkage in a *header file* indicates that they are intended to be accessed from multiple *translation units*. Requiring that this *header file* is included by every *translation unit* that defines or uses the function or object ensures that the declaration matches the definition.

If usage from multiple *translation units* is not required, then the visibility of the function or object should be reduced by declaring it with internal linkage, for example, by declaring it within an unnamed *namespace* of an *implementation file* (see Rule 6.5.2). This has the effect of increasing isolation and encapsulation, which is considered to be good practice.

Compliance with this rule helps to prevent the issues identified in Rule 6.2.2.

Example

```
// header.hpp
extern int32_t a1;        // Compliant

extern void f3();         // Compliant

// file1.cpp
#include "header.hpp"

int32_t a1 = 0;           // Redeclaration - rule does not apply
int32_t a2 = 0;           // Non-compliant - no declaration in header

namespace
{
  int32_t const a3 = 0;   // Internal linkage - rule does not apply

  void f1()               // Internal linkage - rule does not apply
  {
  }
}

void f2()                 // Non-compliant - no declaration in header
{
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f3()                   // Redeclaration - rule does not apply
{
}
```

See also

Rule 6.2.2, Rule 6.5.2

Rule 6.5.2 Internal linkage should be specified appropriately

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

Internal linkage for an *entity* is specified appropriately when:

1. It is declared within an anonymous namespace; and
2. None of its *declarations* use the `extern` keyword; and
3. It is not declared `static`.

This rule does not apply to variables declared `constexpr` or `const`.

Rationale

Whilst the `static` keyword can be used to give an *entity* internal linkage, it also has other uses, which may lead to confusion. An *entity* is unambiguously given internal linkage when it is declared in an anonymous *namespace*, with the added advantage that this declarative form can be consistently applied to all types of *entity*.

An *entity* in an anonymous *namespace* can be declared `extern`, but this does not have an impact on its linkage.

Example

```
static void f1();               // Non-compliant

namespace
{
  void f2();                    // Compliant

         int32_t notExtern1;    // Compliant
  extern int32_t notExtern2;    // Non-compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

4.6.7 Storage duration [basic.stc]

Rule 6.7.1 Local variables shall not have static storage duration

[basic.stc.static]
[basic.start.dynamic]
[basic.start.term]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to variables declared `constexpr` or `const`.

Rationale

The use of mutable variables with static storage duration, even when they do not have linkage, potentially results in hidden temporal coupling. This can lead to data races (and thus *undefined behaviour*). Additionally, functions with persistent state are usually more difficult to understand and test.

Note: the lifetime of local variables with static storage duration ends at program termination in the reverse order of their creation. Suitable care should be taken to ensure that the code executed during destruction does not access a previously destroyed variable.

Example

```
int32_t bar();

int32_t ga = 0;                         // Compliant - but violates "See also"

int32_t foo()
{
                    int32_t a = 0;       // Compliant
  static            int32_t b = 0;       // Non-compliant
  static constexpr int32_t c = 0;        // Compliant
  static const      int32_t d = bar();   // Compliant
}

class Application
{
  static Application & theApp()
  {
    static Application app;             // Non-compliant

    return app;
  }
};
```

See also

Rule 6.7.2

Licensed to: gunjanjain
22 Apr 2025

Rule 6.7.2 *Global variables* shall not be used

[basic.stc.static]
[basic.start.dynamic]
[basic.start.term]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Global variables are:

1. Variables defined in namespace scope; and
2. Class static data members.

This rule does not apply to *global variables* that are:

1. `constexpr`; or
2. `const` and that are initialized through *static initialization*.

Rationale

Global variables can be accessed and modified from:

- Anywhere within the *translation unit*, if they have internal linkage; or
- Anywhere within the program, if they have external linkage.

This can lead to uncontrollable interactions between functions, with the risk of *undefined behaviour* occurring due to data races in concurrent programs.

Additionally, certain aspects of the order of initialization of global variables are *unspecified*. This can lead to unpredictable results for global variables that are initialized at run-time (*dynamic initialization*).

Example

```
int32_t foo();

      int32_t i1 { foo() };                // Non-compliant
const int32_t i2 { i1 };                   // Non-compliant - dynamic initialization

namespace
{
  int32_t i3 { 0 };                        // Non-compliant

  constexpr int32_t bar()
  {
    return 42;
  }

  constexpr int32_t i4 { bar() };          // Rule does not apply - constexpr

  const int32_t SIZE { 100 };              // Rule does not apply
}                                          // - const without dynamic initialization

struct ComplexInit
{
  ComplexInit();
};

const ComplexInit c1 {};                   // Non-compliant - dynamic initialization
```


Licensed to: gunjanjain
22 Apr 2025

```
class StaticMember
{
        int32_t x;                      // Rule does not apply
  static int32_t numInstances;
};

int32_t StaticMember::numInstances = 0;   // Non-compliant

constexpr auto add =                    // Rule does not apply - add is const
  []( auto x, auto y ) { return x + y; };
```

See also

Rule 6.7.1

4.6.8 Object lifetime [basic.life]

Rule 6.8.1 An object shall not be accessed outside of its lifetime

[basic.life] Undefined 5
[class.union]

Category Required

Analysis Undecidable, System

Amplification

Technically, a C++ object does not exist outside of its lifetime. However, for the purposes of this rule, a violation occurs whenever a memory location that does not contain a live object of an appropriate type is accessed.

Rationale

It is *undefined behaviour* to access an object before it has been initialized or after its lifetime has ended (even if the memory occupied by the object has not been released).

It is possible to refer to an object before its lifetime has begun, for example, by referring to a non-active member of a union.

Compliance with the rules cross-referenced by this rule helps to prevent lifetime violations.

Example

```
struct X
{
  void doSomething() {}
};

void h( X * px )
{
  px->~X();                         // End the lifetime of *px
  px->doSomething();                // Non-compliant
}

void g()
{
  X a{};
  auto & b = ( X{} = a );           // Immediate dangling of b
  b.doSomething();                  // Non-compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f()
{
  int32_t * pi = new int32_t { 42 };

  delete pi;
  std::cout << *pi;                    // Non-compliant
}

union u
{
  int32_t  a;
  uint16_t b[ 2 ];
};

uint16_t u2()
{
  u o;

  o.a = 42;
  return o.b[ 0 ];                     // Non-compliant - b is not the active member
}
```

See the cross-referenced rules for further examples.

See also

Rule 6.7.2, Rule 6.8.2, Rule 6.8.3, Rule 6.8.4, Rule 9.5.2, Rule 12.3.1, Rule 18.3.3

Rule 6.8.2 A function must not return a reference or a pointer to a local variable with automatic storage duration

Category Mandatory

Analysis Decidable, Single Translation Unit

Amplification

This rule also applies to:

1. Function parameters passed by value; and
2. Returning a lambda that captures by reference a variable with automatic storage duration; and
3. Returning a lambda that captures the address of a variable with automatic storage duration.

For the purposes of this rule, a `throw` that is not caught within the function is considered to be a return from the function.

Rationale

Automatic variables are destroyed when a function returns. Returning a reference or pointer to such a variable allows it to be used after its destruction, leading to *undefined behaviour*.

Note: this rule and Rule 6.8.3 use decidable checks that allow trivial, specific instances of potentially dangling references to be detected statically. Other (possibly non-decidable) cases are covered by Rule 6.8.1.

Licensed to: gunjanjain
22 Apr 2025

Example

```
int32_t * f1()
{
  int32_t x = 99;

  return &x;                  // Non-compliant
}

int32_t * f2( int32_t y )
{
  return &y;                  // Non-compliant
}

int32_t & f3()
{
  int32_t x = 99;

  return x;                   // Non-compliant
}

int32_t & f4( int32_t y )
{
   return y;                  // Non-compliant
}

int32_t & f5( int32_t & x )
{
  return x;                   // Rule does not apply
}

int32_t * f6()
{
  static int32_t x = 0;

  return &x;                  // Rule does not apply
}

void f7()
{
  int32_t x = 0;

  throw &x;                   // Non-compliant
}

void f8()
{
  try
  {
    int32_t x = 0;

    throw &x;                 // Rule does not apply - caught within this function
  }
  catch ( ... )
  {
  }
}

auto f9()
{
  int32_t x { 42 };

  return [&x]() {};           // Non-compliant - captures local by reference
}
```


Licensed to: gunjanjain
22 Apr 2025

```
auto f10()
{
  int32_t x { 42 };

  return [p = &x]() {};      // Non-compliant - captures address of local
}
```

The following example is compliant with this rule, but violates Rule 6.8.1.

```
int32_t * f11()
{
  int32_t   i = 42;
  int32_t * p = &i;

  return p;                  // Compliant with this rule
}
```

See also

Rule 6.8.1, Rule 6.8.3

Rule 6.8.3 An assignment operator shall not assign the address of an object with automatic storage duration to an object with a greater lifetime

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies when the right-hand side of an assignment operator has the form `addressof( x )`, `&x`, or is the name of an object having array type.

For the purposes of this rule, two objects with automatic storage duration that are declared in the same scope are considered to have the same lifetime.

Rationale

If the address of an automatic object is assigned to another automatic object of larger scope, or to an object with static storage duration, then the object containing the address may exist beyond the time when the original object ceases to exist (and its address becomes invalid).

Note: this rule and Rule 6.8.2 use decidable checks that allow trivial, specific instances of potentially dangling references to be detected statically. Other (possibly non-decidable) cases are covered by Rule 6.8.1.

Example

```
void f1()
{
  int8_t * p;

  {
    int8_t local;
    int8_t local_array[ 10 ];

    p = &local;                  // Non-compliant
    p = local_array;             // Non-compliant
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

The following example is compliant with this rule, but violates Rule 6.8.1.

```
void f2()
{
  int8_t * p1;

  {
    int8_t * p2 = nullptr;
    int8_t local;

    p2 = &local;                // Compliant - objects have the same lifetime
    p1 = p2;                    // Rule does not apply
  }

  *p1 = 0;                      // Undefined behaviour
}
```

See also

Rule 6.8.1, Rule 6.8.2

Rule 6.8.4 Member functions returning references to their object should be *ref-qualified* appropriately

[basic.life]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to member functions with reference or pointer return type, where, in the definition of the function, at least one return expression explicitly designates `this`, `*this` or a subobject of `*this`.

Such a function is only appropriately *ref-qualified* when:

1. It is *non-const-lvalue-ref-qualified* (`&`); or
2. It is *const-lvalue-ref-qualified* (`const &`) and another overload of that function is declared that is *rvalue-ref-qualified* (`&&`) with the same *parameter-type-list*.

Note: this implies that a member function returning a pointer or reference to its object should be *ref-qualified*, but not *rvalue-ref-qualified*.

Rationale

Returning a reference or pointer to a temporary object, or one of its subobjects, from a member function can lead to immediate dangling.

Ref-qualification of member functions can be used to control which of them can be called on a temporary object:

1. A *non-const-lvalue-ref-qualified* function will never bind to a temporary object; and
2. A *const-lvalue-ref-qualified* function could bind to a temporary object, but this will not occur if an *rvalue-ref-qualified* overload is present as it will be preferred during overload resolution.

Compliance with this rule ensures that member functions directly returning references to their object members cannot be called on temporary objects. This rule is limited to direct references so that checks for compliance are decidable. Use of an indirect reference to a temporary object after its lifetime has ended is covered by Rule 6.8.1.

Licensed to: gunjanjain
22 Apr 2025

Notes:

1. An *rvalue-ref-qualified* member function will only bind to temporary objects and should therefore never return a reference or pointer to its object or one of its subobjects.
2. This rule does not apply to defaulted assignment operators as they do not have a definition. However, Rule 8.18.2 prevents the implicitly returned reference from being used.

Example

```
struct A
{
  int32_t   a;                        // a is a subobject of *this
  int32_t & b;                        // b is a reference, not a subobject of *this

  int32_t & geta1() &                 // Compliant - non-const-lvalue-ref-qualified
  { return a; }

  int32_t const & geta2() const &     // Compliant - const-lvalue-ref-qualified and
  { return a; }

  int32_t geta2() &&                  //             this rvalue-ref overload exists
  { return a; }

  int32_t & getb()                    // Rule does not apply - b is not a subobject
  { return b; }

  A const * getMe1() const & { return this; }  // Compliant
       void getMe1() const && = delete;        //  - because this overload exists

  A & getMe2() { return *this; }               // Non-compliant - not ref-qualified
};

A createA();

// A call to the non-compliant getMe2 on a temporary results in immediate dangling
A & dangling = createA().getMe2();
```

This rule does not apply to the following example, which is still dangerous and could lead to the use of a dangling pointer (see Rule 6.8.1):

```
class C
{
  C * f()
  {
    C * me = this;

    return me;                                 // Indirectly designates 'this'
  }
};
```

In the following example, the instantiation of `f` in the call at `#2` is compliant because `#1` is an overload of `f` with the same *parameter-type-list*. However, the instantiation of `f` in the call at `#3` does not have such an overload and is therefore non-compliant.

```
struct Tmpl
{
  template< typename T >
  Tmpl const * f( T ) const & { return this; } // Non-compliant when instantiated
                                               // for #3
  void f( int32_t ) const && = delete;         // #1
};
```


Licensed to: gunjanjain
22 Apr 2025

```
void bar( int32_t s32, int8_t s8 )
{
  Tmpl tpl;

  tpl.f( s32 );                                  // #2
  tpl.f( s8  );                                  // #3
}
```

See also

Rule 6.8.1, Rule 8.18.2, Rule 15.0.2

4.6.9 Types [basic.types]

Rule 6.9.1 The same type aliases shall be used in all *declarations* of the same *entity*

Category Required

Analysis Decidable, Single Translation Unit

Rationale

If a *redeclaration* uses different type aliases to those in its previous *declarations*, it may not be clear that the *declarations* refer to the same *entity*.

Example

```
typedef int32_t INT;
using Index = int32_t;

       INT     i;
extern int32_t i;             // Non-compliant

       INT j;
extern INT j;                 // Compliant
```

In the following, there are two *declarations* of `g`, even though the types differ due to the top level `const` qualifier:

```
void g( int32_t     i );
void g( Index const i );        // Non-compliant - int32_t vs. Index

void h( Index         i     );
void h( Index const   index );  // Compliant - Index used consistently
void h( int32_t     * i     );  // Rule does not apply - different overload
```

See also

Rule 6.2.2

Licensed to: gunjanjain
22 Apr 2025

Rule 6.9.2 The names of the *standard signed integer types* and *standard unsigned integer types* should not be used

[basic.fundamental] Implementation 1, 5

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the names of integral types constructed using the keywords `char`, `short`, `int`, `long`, `signed` and `unsigned` (ignoring any *cv-qualification*). It does not apply to the use of plain `char`.

Rationale

It is *implementation-defined* how much storage is required for any object of the *standard signed integer types* or *standard unsigned integer types*. When the amount of storage being used is important, the use of types having specified widths makes it clear how much storage is being reserved for each object.

The C++ Standard Library *header file* `<cstdint>` often provides a suitable set of integer types having specified widths. If a project defines its own type aliases, it is good practice to use `static_assert` to validate any size assumptions. For example:

```
using torque_t = unsigned short;

static_assert( sizeof( torque_t ) >= 2 );
```

Notes:

1. Compliance with this rule does not prevent integer promotion, which is influenced by the implemented size of `int` and the type used for an alias. For example, an expression of type `int16_t` will only be promoted if the aliased type has a rank lower than that of `int`. The presence or absence of integer promotion may have an influence on overload resolution.
2. Strong typing, which may be provided by `class` or `enum` types, is more robust than the use of type aliases to represent specific width types.

Exception

1. The names of the *standard signed integer types* and *standard unsigned integer types* may be used to define a type alias.
2. The name `int` may be used for:
   a. The parameter to a postfix operator, which **must** use that type; and
   b. The return type of `main`; and
   c. The `argc` parameter to `main`.

Example

```
#include <cstdint>

int           x = 0;                    // Non-compliant - use of int
int32_t       y = 0;                    // Compliant
int_least32_t z = 0;                    // Compliant

using torque_t = int;                   // Compliant by exception #1
torque_t w = 0;
```


Licensed to: gunjanjain
22 Apr 2025

```
class C
{
public:
  C  operator++( int );                    // Compliant by exception #2.1
};

int main() { }                             // Compliant by exception #2.2
int main( int argc, char * argv[] ) { }    // Compliant by exception #2.2 and #2.3
```

4.7 Standard conversions

4.7.0 The built-in type rules [misra]

4.7.0.1 Motivation

The C++ built-in operators perform many implicit conversions on their operands. These conversions can lead to unexpected information loss, change of signedness, *implementation-defined behaviour* or *undefined behaviour*. To mitigate this, the guidelines in this section, collectively called the *built-in type rules*, place restrictions on expressions that contain instances of *integral promotion* and the *usual arithmetic conversions*.

Note: the *built-in type rules* do not prevent all possible *undefined behaviour* (e.g. signed integer overflow) related to the use of the built-in operators.

For the following reasons, the *built-in type rules* are stricter than the related guidelines in MISRA C, even though the behaviour of both languages is similar:

1. C++ allows function overloading, and knowing which overload is selected by overload resolution requires a clear understanding of a sub-expression's type.
2. C++ allows the definition of user-defined types that wrap and mimic numeric types. In addition to the benefits of strong typing, such types can be written in a way that prevents dangerous, implicit conversions. These types can be used when arithmetic with small types is required in a program, meaning that the *built-in type rules* can be very strict without harming the expressivity of the developer.

4.7.0.2 Scope and definitions

Type categories

Every built-in type is assigned a *type category*, as follows:

Type category	Types
Character category	`char`, `wchar_t`, `char16_t`, and `char32_t`
Integral category	*signed integer types* and *unsigned integer types*, including `signed char` and `unsigned char`
Floating category	`float`, `double`, and `long double`
Other	`bool`, `void`, and `nullptr_t`

The *integral category* and *floating category* types are collectively called the *numeric types*.

For the purposes of the *built-in type rules*:

1. It is assumed that all bits of a *numeric type* participate in its *value representation*.
2. All *cv-qualifiers* and *ref-qualifiers* are ignored (for example `int const &` is treated as `int`).

Licensed to: gunjanjain
22 Apr 2025

3. An expression of type *unscoped enumeration* with a *fixed underlying type* is considered as if its type were the underlying type of the enumeration. For example:
   `enum E : int16_t { a, b }` — considered to be of type `int16_t`.
4. A bit-field is considered as having the smallest integer type with the same signedness that is able to represent all the possible values of the bit-field. For example:
   `uint32_t flag : 1` — considered to be of type `uint8_t`.
5. All type aliases that resolve to the same built-in type are considered to be equivalent.

Note: an expression of type *unscoped enumeration* without a *fixed underlying type* is considered to be a symbolic abstraction — see Rule 10.2.3.

Operators

All references to operators within the *built-in type rules* only refer to built-in operators. An expression that uses an operator that resolves to an overloaded operator is a function call.

For the built-in operators, the following operator categories are defined:

Category	Operators
Arithmetic operators	`+ - * / %`
Bitwise operators	`~ | ^ &`
Shift operators	`>> <<`
Equality operators	`== !=`
Relational operators	`< > <= >=`
Logical operators	`! && ||`
Inc/Dec operators	`++ --`

The restrictions enforced by the *built-in type rules* on the binary forms of the operators in the above also apply to the corresponding compound assignment forms (`%=`, etc.). For instance, `a += b` is considered to be equivalent to `a + b`.

Assignment

For the purposes of the *built-in type rules*, the term *assignment* is not limited to the use of the *assignment operator* and, designates those constructs that may include an implicit conversion between the type of a source expression and a determined target type.

The following are *assignments*:

1. Assigning a value using the *assignment operator*; and
2. Initializing a variable, including within a lambda capture; and
3. Passing a function parameter by value, including passing a default value for a function argument and passing a parameter to a function that is called implicitly (such as the call to a constructor, to overloaded operators, to `operator()` of closure types); and
4. Returning a value from a function by value; and
5. Using a value in a `switch` statement's `case` label (source expression), where the target type is given by the adjusted type of the *condition*.

Note: compound assignments are not *assignments*.

4.7.0.3 Conventions used in examples

For the sake of simplicity, the examples within the *built-in type rules* all assume that `char` is 8 bit, `short` is 16 bit, `int` is 32 bit, `long` is 32 bit and `long long` is 64 bit.

Licensed to: gunjanjain
22 Apr 2025

However, when enforcing the guidelines within a project, it is the sizes of the types provided by the implementation that are taken into account, which means that the compliance of code may depend on the target architecture.

The examples also assume that the following user-defined literal is defined:

```
constexpr uint16_t operator ""_u16( unsigned long long val );
```

Similarly for `_u8`, `_u32`, `_u64`, `_s8`, `_s16`, `_s32` and `_s64`.

Additionally, any variable whose name starts with `u8`, `u16`, `u32`, `u64`, `s8`, `s16`, `s32`, `s64` should be assumed to be of the indicated type. Variables whose names start with `f`, `d` or `ld` are respectively of type `float`, `double` or `long double`.

Rule 7.0.1 There shall be no conversion from type `bool`

[conv.prom]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to all implicit and explicit conversions, except for:

1. The operands of an *equality operator* where both operands have type `bool`; or
2. An explicit cast from `bool` to `class T`, when that class has a converting constructor with a parameter of type `bool`:
   a. `T { true }`
   b. `T ( true )`
   c. `static_cast< T >( true )`
   d. `( T )true` — note that this violates Rule 8.2.2

Rationale

Values of type `bool` may be subject to integral promotion and the usual arithmetic conversions. However, occurrences are generally indicative of an error or code obfuscation. For example, the use of `bool` operands with the *bitwise operators* is unlikely to be intentional and is likely to indicate that a bitwise operator (`&`, `|`, `~`) has been confused with a logical operator (`&&`, `||`, `!`). This rule allows errors such as this to be detected.

The integral promotion that occurs when an *equality operator* is used to compare two values of type `bool` is permitted as it is benign.

Casting a `bool` to an integral type is not allowed as it is clearer to specify the values to which `true` and `false` will be converted.

Exception

As there is no risk of confusion, a value of type `bool` may be *assigned* to a bit-field of length 1 — this is a common idiom used when accessing hardware registers.

Example

```
bool   b1  = true;
bool   b2  = false;
double d1;
int8_t s8a;
```


Licensed to: gunjanjain
22 Apr 2025

```
if ( b1 & b2 )                      // Non-compliant - b1 and b2 converted to int
if ( b1 < b2 )                      // Non-compliant - b1 and b2 converted to int
if ( ~b1 )                          // Non-compliant - b1 converted to int
if ( b1 ^ b2 )                      // Non-compliant - b1 and b2 converted to int
if ( b1 == 0 )                      // Non-compliant - b1 converted to int

double result = d1 * b1;            // Non-compliant - b1 converted to double
s8a = static_cast< int8_t >( b1 );  // Non-compliant - b1 converted to int8_t

if ( b1 == false )                  // Compliant - Boolean operands to equality
if ( b1 == b2 )                     // Compliant - Boolean operands to equality
if ( b1 != b2 )                     // Compliant - Boolean operands to equality
if ( b1 && b2 )                     // Compliant - no conversion from bool
if ( !b1 )                          // Compliant - no conversion from bool

s8a = b1 ? 3 : 7;                   // Compliant - no conversion from bool

void f( int32_t n );
bool b;

f( b );                             // Non-compliant - b converted to int32_t
f( b ? 1 : 0 );                     // Compliant - no conversion from bool

switch ( b )                        // Non-compliant - b converted to int

struct A
{
  explicit A( bool );
};

A anA { true };                     // Compliant - constructor

anA = A { false };                  // Compliant - explicit cast calls constructor
```

See also

Rule 7.0.2, Rule 8.2.2

Rule 7.0.2 There shall be no conversion to type `bool`

[conv.integral]
[conv.bool]

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Conversion from a fundamental type (see [basic.fundamental]) to `bool` depends on the interpretation of a non-zero value as `true` (see [conv.bool]). However, this interpretation may not be appropriate for APIs, such as POSIX, that do not use Boolean return values.

Contextual conversion to bool occurs when an operand of fundamental type is used as:

- An operand to a *logical operator*; or
- The first operand of the conditional operator; or
- The *condition* of an *if-statement* or *iteration-statement*.

The result of such a conversion may not be what the developer intended (for example, when an assignment is accidentally used as the *condition* to an *if-statement*). Therefore, wherever a *contextual conversion to bool* may occur, the corresponding expression shall have type `bool` (e.g. as a result of an explicit test).

Licensed to: gunjanjain
22 Apr 2025

In addition, fundamental types, *unscoped enumeration types*, and pointers will be implicitly converted on *assignment* to `bool`. The result of such implicit conversions may not be what the developer intended.

Exception

1. A `static_cast` to `bool` is permitted for a `class` type having an `explicit operator bool`.
2. *Contextual conversion to bool* is permitted from a pointer, or from a `class` type having an `explicit operator bool`.
3. A bit-field of size 1 can be converted to `bool` — this is a common idiom used when accessing hardware registers and there is no risk of confusion.
4. In a *while*, a *condition* of the form *type-specifier-seq declarator* is not required to have type `bool` as alternative mechanisms for achieving the same effect generally require the scope of objects to be wider than necessary. Note that a similar exception is not needed for the *if* statement, as the `if ( init-statement`$_{opt}$ `condition )` syntax was introduced in C++17.

Example

```
if ( ( u8a < u8b ) && ( u8c < u8d ) )  // Compliant
if ( ( u8a < u8b ) && ( u8c + u8d ) )  // Non-compliant

if ( true && ( u8c < u8d ) )           // Compliant
if ( 1    && ( u8c < u8d ) )           // Non-compliant
if ( u8a  && ( u8c < u8d ) )           // Non-compliant

if ( !0 )                              // Non-compliant
if ( !false )                          // Compliant

s32a =   s16a       ? s32b : s32c;     // Non-compliant
s32a =   b1         ? s32b : s32c;     // Compliant
s32a = ( s16a < 5 ) ? s32b : s32c;     // Compliant

int32_t fn();
bool    fb();

while ( int32_t i1 = fn() )            // Compliant by exception #4
if ( int32_t i2 = fn() )               // Non-compliant
if ( int32_t i3 = fn() ; i3 != 0)      // Compliant version of the above line

while ( std::cin )                     // Compliant by exception #2 - std::istream
                                       //   has explicit operator bool
for ( int32_t x = 10; x; --x )         // Non-compliant

extern int32_t * getptr();

if ( getptr() )                        // Compliant by exception #2 - contextual
                                       //   conversion from pointer to bool
bool b2 = getptr();                    // Non-compliant
bool b3 = getptr() != nullptr;         // Compliant

if ( bool b4 = fb() )                  // Compliant
if ( int32_t i = fn(); i != 0 )        // Compliant
if ( int32_t i = fn(); i )             // Non-compliant - condition has type of 'i'
if ( int32_t i = fn() )                // Non-compliant
if ( u8 )                              // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

The following example illustrates *contextual conversion to bool* with a user-defined `class` type:

```
class C
{
  int32_t x;
public:
  explicit operator bool() const { return x < 0; }
};

void foo( C c )
{
  bool b1 = static_cast< bool >( 4 );  // Non-compliant
  bool b2 = static_cast< bool >( c );  // Compliant by exception #1

  if ( c )                             // Compliant by exception #2 - contextual
  {                                    //   conversion from 'C' to bool
  }
}
```

See also

Rule 7.0.1, Rule 7.11.3, Rule 8.2.4

Rule 7.0.3 The numerical value of a character shall not be used

Category Required

Analysis Decidable, Single Translation Unit

Amplification

There shall be no implicit or explicit conversion to or from an expression with *character category*.

This rule does not apply to *unevaluated operands*, or to the operands of an *equality operator* when both have the same character type.

Rationale

Types in the *character category* are used to represent characters, not integers. The meaning of the integer value of a character type depends on behaviour outside of the program and the C++ language, such as the encoding that is being used.

When other operations on characters are necessary, such as determining if a character is lower case, C++ Standard Library functions provide solutions that are safer than the arithmetic manipulation of character values.

Where the underlying numeric representation of character data is required, such as when generating a hash, appropriate conversion functions are provided by `std::char_traits<>`.

Example

```
char a = 'a';              // Rule does not apply - no conversion
char b = '\r';             // Rule does not apply - no conversion
char c = 10;               // Non-compliant - implicit conversion from int to char

int8_t      d = 'a';       // Non-compliant
uint8_t     e = '\r';      // Non-compliant
signed char f = 11;        // Rule does not apply - type has integral category

using CT = std::char_traits< char >;

char g = L'Æ';             // Non-compliant - conversion between character types
char h = a - '0';          // Non-compliant - promotion to int, conversion to char
```


Licensed to: gunjanjain
22 Apr 2025

```
if ( g && h )              // Non-compliant - two conversions to bool
if ( a != 'q' )            // Rule does not apply - comparing the same types
if ( CT::eq( a, 'q' ) )    // Rule does not apply - no conversion

std::optional< char > o;
if ( o == 'r' )            // Rule does not apply - no conversion

decltype( 's' + 't' ) w;   // Rule does not apply - unevaluated operand

auto i = static_cast< CT::int_type >( 'a' );   // Non-compliant - explicitly
                                               // converted to CT::int_type
auto j = CT::to_int_type( 'a' );               // Rule does not apply
                                               //   - no conversion

if ( ( a >= '0' ) && ( a <= '9' ) )            // Non-compliant - promotion to int
if ( !CT::lt( a, '0' ) && !CT::lt( '9', a ) )  // Compliant version of the above

if ( 0 == std::isdigit( a ) )                  // Non-compliant - conversion to int
if ( std::isdigit( a, std::locale {} ) )       // Compliant version of the above

void f1 ( std::istream & is )
{
  auto i = is.get();

  if ( CT::not_eof( i ) )
  {
    char c1 = i;                               // Non-compliant - int to char
    char c2 = CT::to_char_type( i );           // Compliant version of the above
  }
}
```

Rule 7.0.4 The operands of *bitwise operators* and *shift operators* shall be appropriate

[expr.bit.and]
[expr.or]
[expr.xor]
[expr.shift]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The following shall be of an unsigned type:

- Both operands of the binary *bitwise operators*;
- The left operand of the *shift operators*;
- The operand of the bit *complement operator*.

The right operand of the *shift operators*, shall be:

- Either a non-constant expression with an unsigned type; or
- A *constant expression* with a value between `0` and `sizeof( T ) * CHAR_BIT - 1` (inclusive), where `T` is the type of the left operand (before *integral promotion*).

The requirements of this rule for binary *bitwise operators* also apply to the corresponding compound assignment forms.

Licensed to: gunjanjain
22 Apr 2025

Rationale

Bit-oriented operations may be applied to operands of signed and unsigned type. However, the result is only guaranteed to be defined when the sign bit is not affected.

Unlike most other binary operations, the operands to the *shift operators* do not undergo the *usual arithmetic conversions*. Both operands are still subject to *integral promotion*, with the resulting type being the promoted type of the left operand. Explicitly casting the left-hand operand of the *shift operator* to the intended width allows the reader to reason about the code's correctness without having to consider *integral promotion*.

The following behaviours may occur if operands to a *shift operator* have a signed type:

- Shifting by a negative value results in *undefined behaviour*;
- A left-shift of a signed left operand can result in *undefined behaviour*, even when the value is positive;
- Right-shifting a negative value results in an *implementation-defined* value.

Additionally, *undefined behaviour* occurs when a *shift operator* has a right operand with a value that is greater than or equal to the size in bits of the promoted type of the left operand.

Exception

The left operand of a *shift operator* is permitted to be a non-negative *constant expression* of a signed type `T` (before *integral promotion*) when:

- `T` uses two's complement representation; and
- The right operand is also a *constant expression* with a value between `0` and `sizeof( T ) * CHAR_BIT - 1` (inclusive); and
- No set bit is shifted into or beyond the most significant bit, which is used as the sign bit.

Example

The following examples assume `int` uses 32-bit two's complement representation.

```
1u << u8;                                  // Compliant
1u << 31;                                  // Compliant
1_u8 << 2;                                 // Compliant - but violates other rules
u8 << 2;                                   // Compliant - but violates other rules

s32 << 2;                                  // Non-compliant - left operand is signed
1 << u8;                                   // Non-compliant - left operand is signed
( u8 + u16 ) << 2;                         // Non-compliant - result of + is signed
static_cast< uint16_t >( u8 + u16 ) << 2;  // Compliant

1LL << 31;                                 // Compliant by exception
1 << 30;                                   // Compliant by exception
2 << 30;                                   // Non-compliant - exception does not
                                           //   apply as set bit is shifted too far

u32a | u32b;                               // Compliant
s32a | s32b;                               // Non-compliant - signed operands

~u32;                                      // Compliant
~u8;                                       // Compliant - but violates other rules
~s32;                                      // Non-compliant - signed operand
```

See also

Rule 7.0.5

Licensed to: gunjanjain
22 Apr 2025

Rule 7.0.5 *Integral promotion* and the *usual arithmetic conversions* shall not change the signedness or the *type category* of an operand

[conv.prom]
[conv.fpprom]
[conv.integral]
[conv.double]
[conv.fpint]
[conv.rank]
[expr]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to all expressions (including sub-expressions) of *numeric type*. It also applies within preprocessing directives, with the provision that expressions used in `#if` and `#elif` preprocessor directives always have the type `intmax_t` or `uintmax_t`.

For the *usual arithmetic conversions*, only the final type of an operand is considered. For example, in the expression `u32 + u8`, `u8` is first converted to `signed int` through *integral promotion* before being converted to `uint32_t`; it is the final type of `uint32_t` that is considered by this rule.

This rule does not apply to the *integral promotion* of the operand to the increment or decrement operators.

Rationale

Integral promotion and the *usual arithmetic conversions* are usually value-preserving conversions, and it may therefore appear that they are always safe. However, the signedness of the converted type may, possibly surprisingly, not be the same as the signedness of the operand. For example, when an unsigned type is converted to a signed type, an operation may overflow and trigger *undefined behaviour* instead of wrapping.

The increment and decrement operators convert their results to the type of their operand. This may be a lossy, narrowing conversion, but the usefulness of these operators outweighs this risk.

Exception

1. A compile-time constant with signed integral type that has a non-negative value may be converted to an unsigned type through the *usual arithmetic conversions*.
2. A compile-time constant with integral type may be converted to a floating type.

Example

The following non-compliant examples do not directly pose a problem. However, using their results could lead to surprising or *undefined behaviour*.

```
u8a + u8b                              // Non-compliant - u8a and u8b -> signed int
u8a += u8b                             // Non-compliant - same as u8a + u8b
static_cast< uint32_t >( u8a ) + u8b   // Compliant - u8b -> unsigned int
u8a += static_cast< uint32_t >( u8b )  // Compliant - u8a -> unsigned int

s32 * s8                               // Compliant - s8 -> signed int
u32 / u8                               // Compliant - u8 -> unsigned int
s32 > u32                              // Non-compliant - s32 -> unsigned int
u32a - 1                               // Compliant by exception #1
```


Licensed to: gunjanjain
22 Apr 2025

```
b ? u8a : u8b                          // Compliant - no conversion
b ? u8a : u16a                         // Non-compliant - u8a -> signed int and
                                       //                 u16a -> signed int

array[ u8 ]                            // Rule does not apply - no conversion of u8
u8++                                   // Rule does not apply

f32 += u32                             // Non-compliant - u32 -> floating
f32 += 1                               // Compliant by exception #2
f32 += 0x100'0001                      // Compliant by exception #2 - precision lost

~u8                                    // Non-compliant - u8 -> signed int
~u32                                   // Compliant
-u8                                    // Non-compliant - u8 -> signed int
u8 << 2                                // Non-compliant - u8 -> signed int

constexpr int32_t fn( int32_t i )
{
  return i * i;
}

u8  + fn( 10 )                         // Compliant by exception #1
f32 + fn( 10 )                         // Compliant by exception #2
```

Rule 7.0.6 *Assignment* between numeric types shall be appropriate

[conv]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to all *assignments* where the source and target have *numeric* type.

A call is *non-extensible* when it is:

- A qualified call to a member function (such as `a.f( x )`, `this->f( x )`, or `A::f( x )`); or
- A call to an `operator()`.

A function argument `arg` is *overload-independent* when the call is:

- Through a pointer to function or pointer to member function; or
- *Non-extensible*, and, for all overloads that are callable with the same number of arguments, the parameters corresponding to `arg` have the same type. Note that a parameter of a function template that is dependent on a function template parameter never has the same type.

The source and target within an *assignment* shall have the same type when the source expression is:

1. An argument to a function call (including an implicit constructor call) and the corresponding parameter is not *overload-independent*; or
2. Passed through the ellipsis parameter in a function call (where the target type is the promoted type of the argument).

For all other *assignments*:

1. The source and target shall have types of the same *type category*, signedness and size; or
2. The source and target shall have types of the same *type category*, signedness, the source size shall be smaller than the target size, and the source shall be an *id-expression*; or

Licensed to: gunjanjain
22 Apr 2025

3. The source shall be an integer constant expression and the target shall be either:
   a. Any *numeric* type with a range large enough to represent the value, even if the value is not exactly representable (when storing to a float, for example); or
   b. A bit-field whose value representation width (see [class.bit]/1) and signedness are capable of representing the value.

Rationale

The C++ built-in operators perform many implicit conversions on their operands. These conversions can lead to unexpected information loss, change of signedness, *implementation-defined behaviour* or *undefined behaviour*. This rule therefore places restrictions on the presence of implicit numeric conversions on *assignment*.

For floating-point types, the exact representation of a value is often not possible, so loss of precision when assigning a constant value is not a violation of this rule, provided it is within the range of the target type.

Additionally, implicit conversions on *assignment* to a function parameter are undesirable as they could result in a silent change in overload selection due to changes elsewhere within the code, such as the addition of a `#include`. For this reason, the implicit conversion of a function argument is not permitted — except when the corresponding parameter is *overload-independent*, in which case an implicit conversion of the *type category* is permitted as a silent change in overload selection cannot occur.

Exception

The *assignment* to a parameter within a call to a constructor that is callable with a single *numeric* argument is permitted to have a target type that is a wider version of the source type, provided that the class has no other constructors that are callable with a single argument, apart from copy or move constructors. This allows an instance of the class to be created and used as a function parameter without requiring an explicit widening conversion of the source type.

Example

```
u32 = 1;                        // Compliant
s32 = 4u * 2u;                  // Compliant
u8  = 3u;                       // Compliant
u8  = 3_u8;                     // Compliant
u8  = 300u;                     // Non-compliant - value does not fit
```

The use of bit-fields in the following example violates Rule 12.2.1.

```
struct S { uint32_t b : 2; } s; // Bit-field is considered to be uint8_t

s.b = 2;                        // Compliant
s.b = 32u;                      // Non-compliant - value does not fit
s.b = u8;                       // Compliant - same width, but may truncate
s.b = u16;                      // Non-compliant - narrowing

void sb1( uint32_t );
void sb1( uint8_t  );
void sb2( uint8_t  );

void sb3()
{
  sb1( s.b );                   // Non-compliant - s.b considered to be uint8_t,
                                //                 but sb1( uint32_t ) is called
  sb2( s.b );                   // Compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

```
enum Colour : uint16_t
{
  red, green, blue
} c;

u8  = red;                      // Compliant - value can be represented
u32 = red;                      // Compliant - value can be represented
u8  = c;                        // Non-compliant - different sizes (narrowing)
u32 = c;                        // Compliant - widening of id-expression

enum States
{
  enabled, disabled
};

u8 = enabled;                   // Rule does not apply - source type not numeric

unsigned long ul;
unsigned int  ui = ul;          // Compliant - if sizes are equal

  u8 = s8;                      // Non-compliant - different signedness
  u8 = u8 + u8;                 // Non-compliant - change of sign and narrowing

flt1 = s32;                     // Non-compliant - different type category
flt2 = 0.0;                     // Non-compliant - different sizes and not an
                                //                 integral constant expression
flt3 = 0.0f;                    // Compliant
flt4 = 1;                       // Compliant
flt5 = 9999999999;              // Compliant - loss of precision is possible

int f( int8_t s8 )
{
  int16_t val1 = s8;            // Compliant
  int16_t val2 { s8 };          // Compliant
  int16_t val3 ( s8 );          // Compliant
  int16_t val4 { s8 + s8 };     // Non-compliant - narrowing, as s8 + s8 is int

  switch ( s8 )
  {
    case 1:                     // Compliant
    case 0xFFFF'FFFF'FFFF:      // Non-compliant - value does not fit in int8_t
      return s8;                // Compliant - widening of id-expression
  }

  return s8 + s8;               // Compliant - s8 + s8 is of type int
}
```

The following examples demonstrate the *assignment* to function parameters that are not *overload-independent*:

```
void f1( int64_t i );

f1( s32 + s32 );                // Non-compliant - implicit widening conversion

void f2( int i );

f2( s32 + s64 );                // Non-compliant - implicit narrowing conversion
f2( s16 + s16 );                // Compliant - result of addition is int

struct A
{
  explicit A( int32_t i );
  explicit A( int64_t i );
};

A a { s16 };                    // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

```
void f3( long l );

void ( *fp )( long l) = &f3;

f3( 2 );                        // Non-compliant - implicit conversion from int to
                                // long. Adding a #include would silently change
                                // the selected overload if it added void f3( int )

fp( 2 );                        // Compliant - calling through function pointer is
                                //             overload-independent

struct MyInt
{
  explicit MyInt( int32_t );
           MyInt( int32_t, int32_t );
};

void f4( MyInt );

void bar ( int16_t s )
{
  f4( MyInt { s } );            // Compliant by exception - no need to cast s
  MyInt i { s };                // Compliant by exception - no need to cast s
}

void log( char const * fmt, ... );

void f( uint8_t c )
{
  log( "f( %c )", c );          // Non-compliant - conversion of c from uint8_t
}                               //                 to int
```

In the following example, all overloads of the function `A::set` that can be called with two arguments have the type `size_t` for their first parameter. Therefore, the first parameter in a qualified call to `A::set` is *overload-independent*:

```
struct A
{
  void set( short value );
  void set( size_t index, int value );
  void set( size_t index, std::string value );
  void set( int index, double value ) = delete;   // Not callable
  void g();
};

void f( A & a )
{
  a.set( 42, "answer" );        // Compliant - size_t can represent 42, and it is
}                               // assigned to an overload-independent parameter

void A::g()
{
  set( 42, "answer" );          // Non-compliant - even though this non-qualified
                                // call will only select an overload in the class
}
```


Licensed to: gunjanjain
22 Apr 2025

In the following example, both overloads of the function `B::set` can be called with two arguments, but their first parameters do not have the same type (even if `int` and `long` have the same size). Therefore, the first parameter in a qualified call to `B::set` is not *overload-independent*:

```
struct B
{
  void set(  int index, int value );
  void set( long index, std::string value );
};

void f( B & b )
{
  b.set( 42, "answer" );        // Non-compliant - conversion from int to long not
}                               // allowed as parameter is not overload-independent

struct C
{
  int32_t x;
  int64_t y;
  int64_t z;
};

C c1 {
       s16 + s16,               // Compliant - s16 + s16 is of type int
       s16 + s16,               // Non-compliant - widening from int
       s16                      // Compliant - widening of id-expression
     };

template< typename T >
struct D
{
  void set1( T index, int value );
  void set1( T index, std::string value );

  template< typename S1 > void set2( S1 index, int value );
  template< typename S2 > void set2( S2 index, std::string value );
};

void f( D< size_t > & a )
{
  a.set1( 42, "X" );            // Compliant - size_t is same type
  a.set2< size_t >( 42, "X" );  // Non-compliant - 'S1' is never the same as
}                               //   the specialized type of 'S2' (size_t)
```

4.7.11 Pointer conversions [conv.ptr]

Rule 7.11.1 `nullptr` shall be the only form of the *null-pointer-constant*

[support.types.nullptr] Implementation 2

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Using any integral literal representing zero, including the literal `0` or the macro `NULL`, to represent the *null-pointer-constant* is a violation of this rule.

In addition, the macro `NULL` shall not be used in any other context.

Licensed to: gunjanjain
22 Apr 2025

Rationale

The C++ Standard defines the object `nullptr` as the *null-pointer-constant*.

The literal `0` can also be used to represent a *null-pointer-constant*. However, `0` has type `int`, and its use can lead to unexpected overload resolution. Note that the macro `NULL` expands to `0`.

Note: some library functions provide overloads for `std::nullptr_t` so that they can be selected during overload resolution at compile-time, avoiding the need for a run-time check against `nullptr`.

Example

```
void f1( int32_t * );

void f2()
{
  f1( nullptr );      // Compliant
  f1( 0 );            // Non-compliant - 0 used as the null pointer constant
}
```

The following example shows the selection of an integer overload when `NULL` (which has a value of `0`) is used instead of `nullptr`:

```
void f3( int32_t   );
void f3( int32_t * );

void f4()
{
  f3( NULL );         // Non-compliant - calls the int32_t   overload
  f3( nullptr );      // Compliant     - calls the int32_t * overload
}
```

The following example shows non-compliant uses of `NULL`, where it is not used as the *null-pointer-constant*:

```
#define MYNULL NULL         // Non-compliant

void f5()
{
  int32_t one = NULL + 1;  // Non-compliant - NULL used as an integer

  throw NULL;              // Non-compliant - caught by catch ( int )
}
```

Rule 7.11.2 An array passed as a function argument shall not decay to a pointer

Category Required

Analysis Decidable, Single Translation Unit

Rationale

An object of array type decays to a pointer when it is passed as a function argument. As a consequence, it becomes more difficult to detect array bounds errors as the array's bounds are lost.

If a design requires arrays of different lengths, then measures shall be taken to ensure that the dimensionality is maintained.

Note: Rule 11.3.1 recommends that C-style arrays should not be used as better options are available in C++.

Licensed to: gunjanjain
22 Apr 2025

Exception

Passing a string literal as an argument to a function that expects a pointer to character parameter is permitted, as the literal is guaranteed to end with a sentinel character (of value `0`) which can be used to detect the end of the array.

Example

```
void f1( int32_t   p[ 10 ]    );            // Array will decay to pointer
void f2( int32_t * p          );            // Array will decay to pointer
void f3( int32_t ( &p )[ 10 ] );            // Only accepts arrays of 10 elements

template< size_t N >                        // Accepts arrays of any size, with the
void f4( int32_t ( &p )[ N ] );             // size being automatically deduced

void f5( initializer_list< int32_t > l );
void log( char const * s );
void log( string_view  s );
```

Note: it is also possible to deduce the size of an array argument without changing the function into a template. For example, an intermediate class can be used to wrap the array, with the constructor deducing the size using the same technique as shown in `f4` (above). Arguments can then use this wrapper class, avoiding the array to pointer decay. The `std::span` class that has been introduced in C++20 uses this idiom.

```
void b()
{
  int32_t a[ 10 ];

  f1( a );   // Non-compliant - dimension lost due to array to pointer conversion
  f2( a );   // Non-compliant - dimension lost due to array to pointer conversion
  f3( a );   // Compliant     - dimension of 10 matches that of the parameter
  f4( a );   // Compliant     - dimension deduced to be 10

  f5( { 1, 2 } );                // { 1, 2 } is an initializer_list, not an array

  log( "Hello" );                // Compliant by exception
  char const msg[] = "Hello";
  log( msg );                    // Non-compliant - not a literal

  string_view msg2 = "Hello"sv;  // Compliant by exception - the literal operator
                                 //   has a string literal as an argument
  log( msg2 );                   // msg2 is a string_view, not an array
}
```

See also

Rule 11.3.1

Rule 7.11.3 A conversion from function type to pointer-to-function type shall only occur in appropriate contexts

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A conversion to pointer to function is appropriate when it occurs:

1. Through a `static_cast`; or
2. In an *assignment* to an object with pointer-to-function type.

Licensed to: gunjanjain
22 Apr 2025

Rationale

The use of a function pointer in Boolean contexts may result in a well-formed program that is contrary to developer expectations. For example, if the developer writes `if ( f )`, then it is not clear whether the intent is to test if the address of the function evaluates to `nullptr`, or that a call to the function `f` should be made but the call operator has been unintentionally omitted. The use of the `&` (*address-of*) operator or an explicit conversion with a `static_cast` to a function pointer removes this ambiguity.

Using a function as an operand in an arithmetic expression will trigger pointer decay.

Example

```
extern int * f();

if ( f == nullptr )          // Non-compliant
{
}

if ( &f != nullptr )         // Compliant - no conversion
{
  (f)();                     // Compliant - no conversion
}

std::cout << std::boolalpha  // Compliant - assignment to pointer-to-function type
          << f;              // Non-compliant - assignment is not to
                             // pointer-to-function type

auto x = +f;                 // Non-compliant

void f1( double   );
void f1( uint32_t );

auto selected = static_cast< void(*)( uint32_t ) >( f1 );  // Compliant

auto lam = [](){};
void ( *p )() = lam;         // Compliant
auto x = +lam;               // Non-compliant

if ( lam )                   // Non-compliant
{
}
```

See also

Rule 7.0.2, Rule 8.2.4

Licensed to: gunjanjain
22 Apr 2025

4.8 Expressions

4.8.0 MISRA [misra]

Rule 8.0.1 Parentheses should be used to make the meaning of an expression appropriately explicit

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

The following table is used in the definition of this rule.

Description	Operator or Operand	Ranking
Other	Any operator or operand not listed below	14 (high)
Multiplicative	`* / %`	13
Additive	`+ -`	12
Bitwise shift	`<< >>`	11
Relational	`< > <= >=`	10
Equality	`== !=`	9
Bitwise AND	`&`	8
Bitwise XOR	`^`	7
Bitwise OR	`|`	6
Logical AND	`&&`	5
Logical OR	`||`	4
Conditional	`?:`	3
Assignment	`= *= /= %= += -= <<= >>= &= ^= |=`	2
Throw	`throw`	1
Comma	`,`	0 (low)

The rankings used in this table are chosen to allow a concise description of the rule. They are not necessarily the same as those that might be encountered in the C++ Standard's descriptions of operator precedence.

Notes:

1. Operators having alternative token representations (see [lex.digraph]) have the same ranking as their primary form.
2. The *additive* row does not include unary plus and unary minus, which have rank 14.

An expression is appropriately explicit when:

- Its ranking is 0, 1, 2 or 14; or
- Each operand:
  - Is parenthesized; or
  - Has a ranking of 14; or
  - Has ranking less than or equal to that of the expression.

Additionally, the operand to the *sizeof* operator should be parenthesized.

Licensed to: gunjanjain
22 Apr 2025

For the purposes of this rule, the ranking of an expression is the ranking of the element (operand or operator) at the root of the parse tree for that expression. For a sub-expression, its ranking is that of the element at the "root" of the sub-tree.

For example, using the syntax and precedence rules from the C++ Standard, the parse tree for the non-compliant expression `a << b + c` can be represented as:

```
     <<
    /  \
   a    +
       / \
      b   c
```

The element at the root of this parse tree is `'<<'`, so the expression has ranking 11. The root of the sub-tree for `b + c` is `+`, which has ranking 12.

Rationale

The C++ language has a comparatively large number of operators and their relative precedences are not intuitive. This can lead less experienced developers to make mistakes. Using parentheses to make operator bindings explicit removes the possibility that the developer's expectations are incorrect. It also makes the original developer's intention clear to reviewers or maintainers of the code.

It is recognized that overuse of parentheses can clutter the code and reduce its readability. However, too few parentheses can lead to unintuitive code. This rule tries to achieve a reasonable compromise.

Note: this rule does not require the operands of a comma operator to be parenthesized, even though the result may not meet developer expectation. However, use of the comma operator is not compliant with Rule 8.19.1.

```
x = a, b;   // Parsed as ( x = a ), b
```

Example

The following examples show expressions with a unary or postfix operator whose operands are either *primary-expressions* or expressions whose top-level operators have ranking 14:

```
a[ i ]->n;              // Compliant - no need to write ( a[ i ] )->n
*p++;                   // Compliant - no need to write *( p++ )
sizeof x + y;           // Non-compliant - write either sizeof ( x ) + y
                        // or sizeof ( x + y )
```

The following examples show expressions containing operators of the same ranking:

```
a + b - c + d;          // Compliant
( a + b ) - ( c + d );  // Compliant - produces a different result
```

The following examples show a variety of mixed-operator expressions:

```
x = f ( a + b, c );     // Compliant - no need to write f ( ( a + b ), c )

x = a == b ? a : a - b; // Non-compliant - operands of conditional operator
                        // (ranking 3) are:
                        //   == (ranking 9) needs parentheses
                        //   a  (ranking 14) does not need parentheses
                        //   -  (ranking 12) needs parentheses

x = ( a == b ) ? a : ( a - b );   // Compliant version of previous example
```

Note: the assignment operators in the previous two examples are compliant — the ranking of the assignment operator is less than 3, so its operands do not need parentheses.

```
x = a << b + c;       // Non-compliant - operands of << operator
                      // (ranking 11) are:
                      //  a  (ranking 14) does not need parentheses
                      //  +  (ranking 12) needs parentheses
```


Licensed to: gunjanjain
22 Apr 2025

```
a && b && c;          // Compliant - all operators are the same.
a && b || c;          // Non-compliant - || (ranking 4) has operand && (ranking 5)
a || b && c;          // Non-compliant - || (ranking 4) has operand && (ranking 5)
a || b || c;          // Compliant - all operators are the same

#if defined( A ) && defined( B ) || defined( C )  // Non-compliant
```

4.8.1 Primary expressions [expr.prim]

Rule 8.1.1 A non-*transient lambda* shall not implicitly capture `this`

Category Required

Analysis Decidable, Single Translation Unit

Rationale

If a lambda with implicit capture (having `=` or `&` in the capture list) attempts to capture a member variable of a class, what is in fact captured is the `this` pointer. This behaviour can be surprising, and may result in *undefined behaviour* if the lambda is called after the object's lifetime has ended. This issue cannot occur for a *transient lambda*.

Note: implicitly capturing `this` using `[=]` is deprecated from C++20.

Example

```
class A
{
  int16_t val;

  void f()
  {
    auto a1 = [=]()                 // Non-compliant - val is not captured, but
              { return val; };      // 'this' is implicitly captured
    auto a2 = [&]()                 // Non-compliant - val is not captured, but
              { return val; };      // 'this' is implicitly captured
    auto a3 = [this]()              // Compliant - 'this' explicitly captured
              { return val; };
    auto a4 = [self = *this]()      // Compliant - current object captured by copy
              { return self.val; };
    auto i  = [&]()                 // Rule does not apply - transient lambda
              { return val; } ();
  }
};
```

Rule 8.1.2 Variables should be captured explicitly in a non-*transient lambda*

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to capture by value and capture by reference.

Rationale

Naming the variables captured by a lambda expression clarifies its dependencies. This allows variables captured by reference and pointers captured by value to be more easily identified, helping to ensure that they are not dangling when the lambda is called.

Licensed to: gunjanjain
22 Apr 2025

This issue cannot occur for a *transient lambda*, so there is no need to explicitly capture its variables.

Example

```
void bar( double val, double min, double max )
{
  auto const easedVal = [&]()
  {
    if ( val < min ) { return ( val + min ) / 2; }
    if ( val > max ) { return ( val + max ) / 2; }
    return val;
  }();                                          // Compliant - called immediately

  auto const ease = [&]()
  {
    if ( val < min ) { return ( val + min ) / 2; }
    if ( val > max ) { return ( val + max ) / 2; }
    return val;
  };                                            // Non-compliant
  ease();                                       //   - not an immediate call
}

template< typename It, typename Func >
bool f1( It b, It e, Func f )                   // f1 does not store f
{
  for ( It it = b; it != e; ++it )
  {
    if ( f( *it ) )                             // f is called
    {
      return true;
    }
  }

  return false;
}

template< typename Cont, typename Func >
bool f2( Cont const & c, Func f )               // f2 does not store f
{
  return f1( std::begin(c), std::end(c), f );   // f passed to non-storing function
}

void foo( std::vector< size_t > const & v, size_t i )
{
  bool b1 = f1( v.cbegin(), v.cend(),
                [&]( size_t elem ) { return elem == i; } );  // Compliant
  bool b2 = f2( v,
                [&]( size_t elem ) { return elem == i; } );  // Compliant
}

struct Speedometer
{
  std::vector< std::function< void ( double ) > > observers;

  template< typename Func >
  void addObserver( Func f )                    // addObserver stores f
  {
    observers.push_back( f );                   // Copying f to the std::function
  }
};

void process( std::function< Speedometer() > );
```


Licensed to: gunjanjain
22 Apr 2025

```
auto f3()
{
  Speedometer s;

  process( [&](){ return s; } );                // Non-compliant - conversion to
                                                //  std::function stores the lambda
  return [=]() { return s; };                   // Non-compliant - implicit capture
}

void addLoggers( Speedometer s, std::ostream & os )
{
  s.addObserver( [&]( double speed )            // Non-compliant - implicit capture
                 { os << speed; });
  s.addObserver( [&os]( double speed )          // Compliant - explicit capture
                 { os << speed; } );
  s.addObserver( []( double speed )             // Compliant - no capture
                 { std::cout << speed; } );
}
```

4.8.2 Postfix expressions [expr.post]

Rule 8.2.1 A virtual base class shall only be cast to a derived class by means of `dynamic_cast`

[expr.static.cast] / 11

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to both pointer and reference casts.

Rationale

The behaviour when casting from a virtual base class to a derived class is only well defined when `dynamic_cast` is used, whilst the use of the other casts can result in *undefined behaviour*. Since C++17, a `static_cast` from a virtual base class is now *ill-formed*, but some compilers may not yet issue a diagnostic. This rule ensures that all cases are detected.

Example

```
class B  { };
class D: public virtual B { };

D d;
B * pB  = &d;
D * pD1 = reinterpret_cast< D * >( pB  );   // Non-compliant
D * pD2 = dynamic_cast< D * >( pB  );       // Compliant - pD2 may be null
D & D3  = dynamic_cast< D & >( *pB );       // Compliant - may throw an exception
```


Licensed to: gunjanjain
22 Apr 2025

Rule 8.2.2 C-style casts and *functional notation* casts shall not be used

[expr.type.conv]
[expr.cast]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to *functional notation* casts that use curly braces or that result in a constructor call.

Rationale

C-style casts and *functional notation* casts raise several concerns:

1. They permit almost any type to be converted to almost any other type without checks;
2. They give no indication why the conversion is taking place;
3. Their syntax is more difficult to recognize.

These concerns can be addressed with the use of `const_cast`, `dynamic_cast`, `static_cast` and `reinterpret_cast`, which:

1. Enforce constraints on the types involved;
2. Give a better indication of the cast's intent;
3. Are easy to identify.

Exception

A C-style cast to `void` is permitted, as this allows the intentional discarding of a value to be made explicit — for instance, the return value of a non-void function call (see Rule 0.1.2).

Example

```
int32_t g();

void f1()
{
  ( void ) g();  // Compliant by exception
}
```

In the following example (which violates Rule 8.2.3), the C-style casts from `a1` to the non-const pointer `a2` is more permissive than necessary. If the type of `a1` is not `A`, then the C-style cast to `a2` will compile, resulting in *undefined behaviour*. The equivalent `const_cast` to `a3` will not compile if the type of `a1` is changed.

```
struct A
{
  A( char c);
};

struct B {};

void f1a( A x )
{
  auto const & a1 = x;
  A          * a2 = ( A * )&a1;                // Non-compliant
  A          * a3 = const_cast< A * >( &a1 );
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f1b( B x )
{
  auto const & a1 = x;
  A          * a2 = ( A * )&a1;                // Non-compliant
  A          * a3 = const_cast< A * >( &a1 );  // Ill-formed
}

void f2( int32_t x )
{
  auto i = A( 'c' );          // Rule does not apply - constructor is called
  auto j = int8_t { 42 };     // Rule does not apply - use of curly braces
  auto k = int8_t ( x );      // Non-compliant - does not construct an object
}                             // of class type
```

See also

Rule 0.1.2, Rule 8.2.3

Rule 8.2.3 A cast shall not remove any `const` or `volatile` qualification from the type accessed via a pointer or by reference

[expr.const.cast] Undefined 1

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Using a cast to remove the qualification associated with the addressed type is a violation of the principle of type qualification.

Some of the problems that might arise include:

- Removal of `const` qualification might circumvent the read-only status of an object, which may lead to *undefined behaviour*;
- Removal of `const` qualification might result in a hardware exception when the object is accessed;
- Removal of `volatile` qualification might result in accesses to an object being removed during optimization.

Example

```
         uint16_t           x;
         uint16_t * const   cpi = &x;       // const pointer
         uint16_t * const * pcpi;           // pointer to const pointer
         uint16_t *       * ppi;
   const uint16_t         * pci;            // pointer to const
volatile uint16_t         * pvi;            // pointer to volatile
         uint16_t         * pi;

pi = cpi;                                   // Rule does not apply - no cast

pi  = const_cast< uint16_t *  >( pci  );    // Non-compliant
pi  = const_cast< uint16_t *  >( pvi  );    // Non-compliant
ppi = const_cast< uint16_t ** >( pcpi );    // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

The following examples also violate Rule 8.2.2.

```
pi  = ( uint16_t *  )pci;                    // Non-compliant
pi  = ( uint16_t *  )pvi;                    // Non-compliant
ppi = ( uint16_t ** )pcpi;                   // Non-compliant
```

See also

Rule 8.2.2

Rule 8.2.4 Casts shall not be performed between a pointer to function and any other type

[expr.reinterpret.cast] Unspecified 6

Category Required

Analysis Decidable, Single Translation Unit

Amplification

For the purposes of this rule, a pointer to a member is considered to be a pointer to function.

The following standard conversions are permitted by this rule, even if they are the result of a cast:

- Function-to-pointer conversions (implicitly taking the address of a function); and
- Function pointer conversions (from a *pointer to noexcept function* to *pointer to function*); and
- Null pointer conversions (from `nullptr` to a *pointer to function*); and
- User-defined conversions, including converting from a lambda with no capture to a *pointer to function*.

Note: the cast notation that is used to disambiguate an overloaded function name ([over.over]) is compliant with this rule because the target type is a function type that is compatible with the source type.

Rationale

Converting a pointer to function into or from any of the following may result in *undefined behaviour*:

- Pointer to object;
- Pointer to non-static data member;
- Pointer to an object of incomplete type;
- `void *`.

Calling a function by means of a pointer whose type is not compatible with the called function also results in *undefined behaviour*.

Casts that are equivalent to a standard conversion cannot lead to those problems and are therefore permitted.

Note: this rule also applies to pointer to member objects as they are *callable* and can be used with `std::invoke`.

Exception

A cast to `void` may be used to signify that a function pointer returned by a function call is being intentionally discarded (see Rule 0.1.2).

Licensed to: gunjanjain
22 Apr 2025

Example

```
using pf16_t = void (*)( int16_t n );
using pf32_t = void (*)( int32_t n );

pf16_t getPf16();

pf16_t p1 = static_cast< pf16_t >( nullptr );   // Compliant - cast is equivalent
                                                // to a standard conversion
pf32_t p2 = reinterpret_cast< pf32_t >( p1 );   // Non-compliant - function pointer
                                                // types are different

( void ) getPf16();                             // Compliant by exception

if ( p1 )                                       // Rule does not apply - no cast;
{                                               // contextually converted to bool
}

pf16_t    p3 = ( pf16_t ) 0x8000;                         // Non-compliant
pf16_t    p4 = reinterpret_cast< pf16_t >( 0xdeadbeef );  // Non-compliant
int16_t * p5 = reinterpret_cast< int16_t * >( p4 );       // Non-compliant

void f5();
void f5( int16_t );

template< typename T >
void f6( T );

void f7()
{
  f6( static_cast< void (*)() >( f5 ) );        // Compliant - overload selection
}

struct A { void foo(); int32_t i; };
struct B : A { };

auto pm1 = static_cast< void ( B::* )() >( &A::foo );      // Non-compliant
auto pm2 = static_cast< int32_t ( B::* ) >( &A::i );       // Non-compliant
```

See also

Rule 0.1.2, Rule 8.2.5

Rule 8.2.5 `reinterpret_cast` shall not be used

[basic.types] / 2
[basic.compound] / 4
[expr.reinterpret.cast]

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Casting between unrelated types generally results in *undefined behaviour*.

Exception

The following are allowed by exception as the behaviour is well defined:

1. Using `reinterpret_cast< T * >` to cast any object pointer to a pointer to `T`, where `T` is one of `void`, `char`, `unsigned char` or `std::byte`, possibly *cv-qualified*.

Licensed to: gunjanjain
22 Apr 2025

2. Using `reinterpret_cast< T >( p )` to convert a pointer `p` to an integer of type `T` that is large enough to represent a pointer value (e.g. `std::uintptr_t`).

Example

```
uint8_t  * p1;
uint32_t * p2;

p2 = reinterpret_cast< uint32_t * >( p1 );                    // Non-compliant

extern uint32_t read_value();
extern void     print( uint32_t n );

void f()
{
  uint32_t   u  = read_value();
  uint16_t * p3 = reinterpret_cast< uint16_t * >( &u );       // Non-compliant
}

void g()
{
  std::array< int32_t, 2 > a{};
  auto p4 = reinterpret_cast< int32_t(*)[ 2 ]>( a.data() );  // Non-compliant

  ( *p4 )[ 0 ] = 42;                                          // Undefined behaviour
}
```

In the following example, the target type `uint64_t` used in the initializer for `p7` violates Rule 8.2.8.

```
void h( float x )
{
  auto p5 = reinterpret_cast< std::byte const * >( &x );  // Compliant by exception
  auto p6 = reinterpret_cast< std::uintptr_t    >( &x );  // Compliant by exception
  auto p7 = reinterpret_cast< uint64_t          >( &x );  // Compliant by exception
}
```

See also

Rule 8.2.1, Rule 8.2.6, Rule 8.2.8

Rule 8.2.6 An object with integral, enumerated, or pointer to `void` type shall not be cast to a pointer type

[expr.reinterpret.cast] Unspecified 7
[expr.static.cast] Undefined 12; Unspecified 13

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply:

- When the destination type is a pointer to function type or a pointer to member function type (see Rule 8.2.4); or
- For casts between pointers to `void`, regardless of any *cv-qualification*.

Rationale

Casting from either an integral type or a pointer to `void` type to a pointer to an object may lead to *unspecified behaviour*.

Licensed to: gunjanjain
22 Apr 2025

A round trip conversion of a pointer to object type through `void *` (`T * -> void * -> T *`) is well-defined. However, this is prohibited by this rule as it is error prone and the detection of any error would be undecidable.

Note: casting from an integer to a pointer may be unavoidable when addressing memory mapped registers or other hardware specific features.

Example

```
struct S
{
  int32_t i;
  int32_t j;
};

void f ( void * p1, int32_t i )
{
     S * s1 = static_cast< S * >( p1 );          // Non-compliant
     S * s2 = reinterpret_cast< S * >( i );      // Non-compliant
  void * p2 = reinterpret_cast< void * >( i );   // Non-compliant
  auto   p3 = const_cast< void const * >( p2 );  // Compliant
}
```

See also

Rule 8.2.4, Rule 8.2.5

Rule 8.2.7 A cast should not convert a pointer type to an integral type

[expr.reinterpret.cast] Implementation 4, 5

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

Casting between a pointer and an integer type makes it harder for tools and developers to understand and reason about code behaviour. For example, pointer tracking within tools may become unreliable when pointers are cast to integers.

Note: casting between pointers and integers may be unavoidable when addressing memory mapped registers or other hardware specific features. When the advice given in this rule is not followed, the use of `std::uintptr_t` or `std::intptr_t` is required by Rule 8.2.8 as these types are guaranteed to be able to represent all possible pointer values.

Example

The following examples violate Rule 8.2.5, with the second also violating Rule 8.2.8:

```
struct S;

void f( S * s )
{
  std::intptr_t p = reinterpret_cast< std::intptr_t >( s );   // Non-compliant
  std::uint8_t  q = reinterpret_cast< std::uint8_t  >( s );   // Non-compliant
}
```

See also

Rule 8.2.5, Rule 8.2.8

Licensed to: gunjanjain
22 Apr 2025

Rule 8.2.8 An *object pointer type* shall not be cast to an integral type other than `std::uintptr_t` or `std::intptr_t`

[expr.reinterpret.cast] Implementation 4, 5
[cstdint.syn]
[basic.compound] / 3

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The *type-id* used in the *cast-expression* shall explicitly specify `std::uintptr_t` or `std::intptr_t`.

Rationale

The types `std::uintptr_t` and `std::intptr_t` are the only types that are guaranteed to be able to represent all possible values of an *object pointer type*.

Note: these types are optional and may not be available in all implementations, in which case a deviation will need to be raised against this rule.

Example

```
struct S;

void f1( S * s )
{
  auto p0 = reinterpret_cast< std::uintptr_t >( s );     // Compliant

  auto p1 = reinterpret_cast< unsigned long >( s );      // Non-compliant

  using hashPtr_t = std::uintptr_t;

  auto p2 = reinterpret_cast< hashPtr_t >( s );          // Non-compliant
}

template< typename T > void f2( S * s )
{
  auto p = reinterpret_cast< T >( s );       // Non-compliant - T is not explicitly
}                                            //                 std::uintptr_t

template void f2< std::uintptr_t >( S * s );
```

See also

Rule 8.2.5, Rule 8.2.7

Licensed to: gunjanjain
22 Apr 2025

Rule 8.2.9 The operand to `typeid` shall not be an expression of polymorphic `class` type

[expr.typeid]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to `typeid( type-id )`.

```
std::type_info const & type { typeid( std::iostream ) };  // Rule does not apply
```

Rationale

An expression of polymorphic `class` type used as the operand to `typeid` may or may not be evaluated at runtime. It is therefore unclear if potential side effects within the expression will or will not occur.

Additionally, `typeid` could throw a `std::bad_typeid` exception, but this will only happen if the operand has polymorphic `class` type.

Note: this rule applies even when there is no runtime evaluation.

Example

```
#include <typeinfo>

struct S { };                          // Non-polymorphic
struct P { virtual void foo() {} };    // Polymorphic

const std::type_info & foo( S * s )
{
  return typeid( *s );                 // Compliant
}

const std::type_info & foo( P * p )
{
  return typeid( *p );                 // Non-compliant
}

const std::type_info & foo( P p )
{
  return typeid( p );                  // Non-compliant
}

const std::type_info & bar( P * p )
{
  return typeid( p->foo() );           // Compliant - type is always 'void'
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 8.2.10 Functions shall not call themselves, either directly or indirectly

Category Required

Analysis Undecidable, System

Rationale

Recursion carries with it the danger of exceeding available stack space, which can lead to a serious failure. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Note: any deviation used to justify non-compliance with this rule will need to explain how stack usage is to be controlled.

Exception

A `constexpr` function that is only called within a *core constant expression* may be recursive.

Example

```
int32_t fn( int32_t x )
{
  if ( x > 0 )
  {
    x = x * fn( x - 1 );              // Non-compliant
  }

  return x;
}

// File1.cpp
int32_t fn_3( int32_t x );

int32_t fn_2( int32_t x )
{
  if ( x > 0 )
  {
    x = x * fn_3( x - 1 );            // Non-compliant
  }

  return x;
}

// File2.cpp
int32_t fn_2( int32_t x );

int32_t fn_3( int32_t x )
{
  if ( x > 0 )
  {
    x = x * fn_2( x - 1 );            // Non-compliant
  }

  return x;
}
```


Licensed to: gunjanjain
22 Apr 2025

In the following, the recursion within `fn_4` satisfies the requirements of the exception as it is **only** called from within a *core constant expression*.

```
constexpr int32_t fn_4( int32_t x )
{
  if ( x > 0 )
  {
    x = x * fn_4( x - 1 );            // Compliant by exception
  }

  return x;
}

constexpr int32_t n = fn_4( 6 );      // Core constant expression

constexpr int32_t fn_5( int32_t x )
{
  if ( x > 0 )
  {
    x = x * fn_5( x - 1 );            // Non-compliant
  }

  return x;
}

int32_t n = fn_5( 6 );                // Not a core constant expression

template< class T >
auto Sum( T t )
{
  return t;
}

template< class T, class ... Vals >
auto Sum( T t, Vals ... vals )
{
  return t + Sum( vals ... );         // Compliant - calls a different overload
}
```

Rule 8.2.11 An argument passed via ellipsis shall have an appropriate type

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The following types are not appropriate:

1. `class` types with virtual member functions;
2. `class` types having non-trivial copy or move operations;
3. `class` types having a non-trivial destructor.

This rule does not apply to unevaluated contexts.

Rationale

Passing arguments of some `class` types via an ellipses parameter is only *conditionally-supported* with *implementation-defined behaviour*.

Licensed to: gunjanjain
22 Apr 2025

Default argument promotions are applied to ellipsis parameters, which may lead to the type that is passed to the function differing from the type that would be passed to a normal function parameter or when passed as a parameter pack.

Note: passing arguments to a parameter pack is not passing via ellipsis.

Example

```
struct A
{
  int i { 42 };
  virtual ~A() = default;
};

void f()
{
  std::printf ( "hello %d", A{} );   // Non-compliant
}
```

The following example uses overload resolution and type deduction; it does not pass an argument via ellipsis at run-time:

```
struct two { char x[2]; };

two  test( int );
char test( ... );

template< typename T >
constexpr bool isIntCompatible( T x )
{
  if constexpr ( sizeof( test( x ) ) == 1 )  // Compliant - unevaluated context
  {
    return false;                            // Overload resolution -> test( ... )
  }

  else
  {
    return true;                             // Overload resolution -> test( int )
  }
}
```

See also

Rule 21.10.1

4.8.3 Unary expressions [expr.unary]

Rule 8.3.1 The built-in unary – operator should not be applied to an expression of unsigned type

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the type of an expression before any integral promotion.

Rationale

Applying the built-in unary – operator to an expression whose promoted type is unsigned generates a result of the same unsigned type, which may not meet developer expectations.

Licensed to: gunjanjain
22 Apr 2025

The result of applying the unary - operator to an expression whose type is unsigned prior to integral promotion will only be negative when the promoted type is signed. The promoted type depends on the operand's rank and the implemented integer sizes.

Example

The following example assumes that `int` is 32 bits.

```
void f(  int32_t a );
void f( uint32_t a );

void g( uint32_t x, uint16_t y )
{
  f( -x );              // Non-compliant - calls f( uint32_t a )
  f( -y );              // Non-compliant - calls f(  int32_t a )
}
```

Rule 8.3.2 The built-in unary + operator should not be used

[expr.unary.op] / 7

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

The built-in unary + operator triggers integral promotion, but otherwise performs no other operation. When its operand is a function name or lambda, decay to a function pointer is triggered. The use of `static_cast` is recommended instead as it makes it clear that these conversions are present.

Example

```
auto  x = + u8a;             // Non-compliant - triggers promotion to int
auto pf = +[](){};           // Non-compliant - pf is a void(*)()

x = +1;                      // Non-compliant
x =+ 1;                      // Non-compliant - unary +, not +=

enum A : uint8_t { one };
enum B : uint8_t { two };

uint8_t operator+( B b ) { return b; }

auto a = +one;               // Non-compliant
auto b = +two;               // Rule does not apply
auto c = operator+( two );   // Rule does not apply
```


Licensed to: gunjanjain
22 Apr 2025

4.8.7 Additive operators [expr.add]

Rule 8.7.1 Pointer arithmetic shall not form an invalid pointer

[expr.add] Undefined 4

Category Required

Analysis Undecidable, System

Amplification

This rule applies to all forms of pointer arithmetic, including array indexing:

```
integer_expression + pointer_expression
pointer_expression + integer_expression
pointer_expression - integer_expression
pointer_expression += integer_expression
pointer_expression -= integer_expression
++pointer_expression
--pointer_expression
pointer_expression++
pointer_expression--
pointer_expression [ integer_expression ]
integer_expression [ pointer_expression ]
```

A pointer resulting from pointer arithmetic is invalid if it does not point to:

- An element of the same array as the original pointer; or
- One past the end of the same array as the original pointer.

This rule also applies to pointer arithmetic that occurs within the C++ Standard Library functions. In addition, it is assumed that the implementation of the functions listed below perform pointer arithmetic on their pointer parameters:

`memchr`, `memcmp`, `memcpy`, `memmove`, `memset`, `strncat`, `strncmp`, `strncpy`, `strxfrm`

Note: a pointer to an object that is not an array is treated as if it were a pointer to the first element of an array with a single element.

Rationale

Undefined behaviour occurs if the result obtained from one of the above expressions is not a pointer to an element of the array pointed to by `pointer_expression`, or a pointer to one beyond the end of that array.

Note: dereferencing an invalid pointer, including a pointer to one past the end of an array, results in *undefined behaviour* — this is targeted by Rule 4.1.3.

Example

```
int32_t * f1( int32_t * const a1, int32_t a2[ 10 ], int32_t ( &a3 )[ 10 ] )
{
  a1[ 3 ] = 0;                  // Compliant only if the array pointed
                                // to by 'a1' has at least 4 elements

  *( a2 + 9 ) = 0;              // Compliant only if the array pointed
                                // to by 'a2' has at least 10 elements

  return a3 + 9;                // Compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f2()
{
  int32_t a1[ 10 ] = { };

  int32_t * p1 = &a1[  0 ];      // Compliant
  int32_t * p2 = a1 + 10;        // Compliant - points to one beyond and
  int32_t    i = *p2;            //             dereferencing is undefined behaviour
  int32_t * p3 = a1 + 11;        // Non-compliant - points to two beyond, resulting
                                 //                 in undefined behaviour

  p1++;                          // Compliant
  a1[ -1 ] = 0;                  // Non-compliant - exceeding array bounds results
                                 //                 in undefined behaviour
  i = *( &i + 0 );               // Compliant     - 'i' is treated as an array
                                 //                 of size 1

  // This declaration has 6 arrays:
  //   1 array of 5 elements of type array of int32_t
  //   5 arrays of 2 elements of type int32_t
  int32_t a2[ 5 ][ 2 ] = { };

  a2[ 3 ][ 1 ] = 0;              // Compliant
  i = *( *( a2 + 3 ) + 1 );      // Compliant
  i = a2[ 2 ][ 3 ];              // Non-compliant - exceeding array bounds results
                                 //                 in undefined behaviour

  int32_t * p4 = a2[ 1 ];        // Compliant

  i = p4[ 1 ];                   // Compliant - p4 addresses an array of size 2
}
```

The following example illustrates pointer arithmetic applied to members of a structure. Because each member is an object in its own right, this rule prevents the use of pointer arithmetic to move from one member to the next.

```
struct
{
  uint16_t x;
  uint16_t y;
  uint16_t a[ 10 ];
} s;

void f3()
{
  uint16_t * p { &s.x };

  ++p;                           // Compliant - p points one past the end of s.x,
                                 //   but this cannot be assumed to point to s.y
  *p = 0;                        //   and dereferencing is undefined behaviour

  ++p;                           // Non-compliant - more than one past the end

  p = &s.a[ 0 ];                 // Compliant     - p points into s.a
  p = p + 8;                     // Compliant     - p still points into s.a
  p = p + 3;                     // Non-compliant - more than one past the end
}
```

The following example shows that the implicit pointer arithmetic within library functions can lead to accesses beyond the end of an array:

```
uint8_t buf1[ 5 ] = { 1, 2, 3, 4, 5 };
uint8_t buf2[ 7 ] = { 1, 2, 3, 4, 5, 6, 7 };

void f4()
{
  if ( std::memcmp( buf1, buf2, 5 ) == 0 ) {}    // Compliant
  if ( std::memcmp( buf1, buf2, 7 ) == 0 ) {}    // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

```
  auto p1 = std::next( buf1, 3 );                // Compliant
  auto p2 = std::next( buf1, 7 );                // Non-compliant
}
```

See also

Rule 4.1.3

Rule 8.7.2 Subtraction between pointers shall only be applied to pointers that address elements of the same array

[expr.add] Undefined 5

Category Required

Analysis Undecidable, System

Amplification

This rule applies to expressions of the form:

`pointer_expression_1 - pointer_expression_2`

Note: a pointer to an object that is not an array is treated as if it were a pointer to the first element of an array with a single element.

Rationale

Undefined behaviour occurs if `pointer_expression_1` and `pointer_expression_2` do not point to elements of the same array or the element one beyond the end of that array.

Example

```
void f1( int32_t * ptr )
{
  int32_t   a1[ 10 ];
  int32_t   a2[ 10 ];
  int32_t * p1 = &a1[ 1 ];
  int32_t * p2 = &a2[ 10 ];

  ptrdiff_t diff1 = p1 - a1;     // Compliant
  ptrdiff_t diff2 = p2 - a2;     // Compliant
  ptrdiff_t diff3 = p1 - p2;     // Non-compliant
  ptrdiff_t diff4 = ptr - p1;    // Non-compliant
}
```

See also

Rule 4.1.3

Licensed to: gunjanjain
22 Apr 2025

4.8.9 Relational operators [expr.rel]

Rule 8.9.1 The built-in relational operators >, >=, < and <= shall not be applied to objects of pointer type, except where they point to elements of the same array

[expr.rel] Unspecified 4

Category Required

Analysis Undecidable, System

Amplification

Uses of `std::less`, `std::less_equal`, `std::greater`, `std::greater_equal` are permitted as specializations for any pointer type yield a strict total order.

Notes:

1. A pointer to one beyond the last element of an array is considered to point to an element of that array.
2. A pointer to an object that is not an array is treated as if it were a pointer to the first element of an array with a single element.

Rationale

Attempting to make comparisons between unrelated pointers may result in surprising or *unspecified behaviour*.

Example

```
void f1()
{
  int32_t   a1[ 10 ];
  int32_t   a2[ 10 ];
  int32_t * p1 = a1;

  if ( p1 < a1 ) {}                    // Compliant
  if ( p1 < std::end( a1 ) ) {}        // Compliant - right operand is one beyond
  if ( p1 < a2 ) {}                    // Non-compliant
  if ( std::less<>{}( p1, a2 ) ) { }   // Compliant
}

struct S
{
  int32_t m1;
  int32_t m2;
};

void f2()
{
  S x { };

  if ( &x.m1 <= &x.m2 ) {}    // Non-compliant - m1 and m2 are not array elements
}
```

See also

Rule 4.1.3

Licensed to: gunjanjain
22 Apr 2025

4.8.14 Logical AND operator [expr.log.and]

Rule 8.14.1 The right-hand operand of a logical && or || operator should not contain *persistent side effects*

Category Advisory

Analysis Undecidable, System

Rationale

The evaluation of the right-hand operand of the && and || operators is conditional on the value of the left-hand operand. If the right-hand operand contains *side effects* then those *side effects* may or may not occur, which may be contrary to developer expectations.

If evaluation of the right-hand operand would produce *side effects* which are not *persistent* at the point in the program where the expression occurs then it does not matter whether the right-hand operand is evaluated or not.

The term *persistent side effect* is defined in Appendix C.

Example

```
uint16_t f( uint16_t y )              // The side effects within f are not
{                                     // persistent, as seen by the caller
  uint16_t temp = y;

  temp = y + 0x8080U;

  return temp;
}

uint16_t h( uint16_t y )
{
  static uint16_t temp = 0;

  temp = y + temp;                    // This side effect is persistent

  return temp;
}

void g( bool ishigh )
{
  if ( ishigh && ( a == f( x ) ) )    // Compliant - f() has no persistent
  {                                   //             side effects
  }

  if ( ishigh && ( a == h( x ) ) )    // Non-compliant - h() has a persistent
  {                                   //                 side effect
  }
}

volatile uint16_t v;
         uint16_t x;

if ( ( x == 0u ) || ( v == 1u ) )     // Non-compliant - access to volatile v
{                                     //                 is persistent
}
```


Licensed to: gunjanjain
22 Apr 2025

```
( fp != nullptr ) && ( *fp )( 0 );     // Non-compliant if fp points to a function
                                       // with persistent side effects
if ( fp != nullptr )
{
  ( *fp )( 0 );                        // Compliant version of the above
}
```

4.8.18 Assignment and compound assignment [expr.ass]

Rule 8.18.1 An object or subobject must not be copied to an overlapping object

[expr.ass] Undefined 8

Category Mandatory

Analysis Undecidable, System

Amplification

This rule applies when:

- A member of a union is copied to a different member of the same union; or
- A slice of an array is copied to an overlapping slice of the same array using `memcpy`.

Rationale

Copying between members of the same union object may result in *undefined behaviour*.

If part of an array is to be copied to another part of the same array (e.g. when moving elements 1 to 10 to elements 0 to 9), then `std::memcpy` may overwrite an element before it has been copied, as there is no guarantee of the order in which they are copied. By contrast, `std::memmove` is guaranteed to handle the overlap appropriately.

Example

The use of unions in the following example is a violation of Rule 12.3.1.

```
void f1( void )
{
  union
  {
    int16_t i;
    int32_t j;
  } a = { 0 };

  a.i = a.i;          // Rule does not apply - same member
  a.j = a.i;          // Non-compliant
}

void f2( std::array< int16_t, 20 > & a )
{
  memcpy ( &a[ 0 ], &a[ 1 ], 10u * sizeof ( a[ 0 ] ) );  // Non-compliant
  memmove( &a[ 0 ], &a[ 1 ], 10u * sizeof ( a[ 0 ] ) );  // Rule does not apply
  memcpy ( &a[ 1 ], &a[ 0 ], 10u * sizeof ( a[ 0 ] ) );  // Non-compliant
  memmove( &a[ 1 ], &a[ 0 ], 10u * sizeof ( a[ 0 ] ) );  // Rule does not apply
  memcpy ( &a[ 0 ], &a[ 5 ],  5u * sizeof ( a[ 0 ] ) );  // Compliant - no overlap
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 8.18.2 The result of an assignment operator should not be *used*

[expr.ass]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to simple and compound assignments, built-in or overloaded, even if they occur in an *infeasible path*. It does not apply to assignments within an *unevaluated operand*.

Rationale

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement, making it more difficult to avoid the *undefined behaviour* covered by Rule 4.6.1.

Example

```
x = y;                      // Compliant
a[ x ] = a[ x = y ];        // Non-compliant - value of x = y is used

if ( bool_a = bool_b )      // Non-compliant - value of bool_a = bool_b is used
{                           // (bool_a == bool_b was probably intended)
}

if ( uint8_t i = y )        // Rule does not apply - not an assignment operator
{
}

if ( ( 0u == 0u ) || ( bool_a = bool_b ) )  // Non-compliant - even though
{                                           // bool_a = bool_b is not evaluated
}

if ( ( x = f() ) != 0 )     // Non-compliant - value of x = f() is used
{
}

a[ b += c ] = a[ b ];       // Non-compliant - value of b += c is used

a = b = c = 0;              // Non-compliant - values of c = 0, b = c = 0 are used
```

See also

Rule 4.6.1

Licensed to: gunjanjain
22 Apr 2025

4.8.19 Comma operator [expr.comma]

Rule 8.19.1 The comma operator should not be used

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the use of the comma operator, but not when a comma is used as the fold operator within a fold expression.

Rationale

Use of the comma operator is generally detrimental to the readability of code, and the same effect can usually be achieved by other means.

Example

```
f( ( 1, 2 ), 3 );                        // Non-compliant - how many parameters?

template< typename ... Ts >
void print_all_of( const Ts &... ts )
{
  ( print( ts ), ... );                  // Rule does not apply
}
```

The following example is non-compliant with other rules:

```
for ( i = 0, p = &a[ 0 ]; i < N; ++i, ++p )    // Non-compliant
{
}
```

4.8.20 Constant expressions [expr.const]

Rule 8.20.1 An unsigned arithmetic operation with constant operands should not wrap

[expr.const]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to any built-in arithmetic operation resulting in an unsigned integral type, where all operands are *constant expressions*.

This rule does not apply to an expression that is not evaluated, for example, because it appears in the right operand of a logical `&&` operator whose left operand is `false` at compile time.

Rationale

Unsigned integer expressions do not overflow, but instead wrap around in a modular way. Any constant unsigned integer expression that wraps will not be diagnosed by the compiler. There may be good reasons to rely on the modular arithmetic provided by unsigned integer types, but the reasons

Licensed to: gunjanjain
22 Apr 2025

are less obvious if wrapping occurs when an operator has constant operands — this may indicate a programming error.

Example

Any unsigned wrapping that occurs during the evaluation of a `case` expression is unlikely to be intentional. In the following example, any value of `BASE` greater than or equal to 65024 would result in wrapping on a machine with a 16-bit `int` type.

```
#define BASE 65024u

switch ( x )
{
  case BASE + 0u:   f(); break;
  case BASE + 1u:   g(); break;
  case BASE + 512u: h(); break;    // Non-compliant - wraps to 0
}
```

In the following example, the expression `DELAY + WIDTH` has the value 70,000, but this will wrap to 4,464 on a machine with a 16-bit `int` type.

```
constexpr auto DELAY { 10000u };
constexpr auto WIDTH { 60000u };

void fixed_pulse()
{
  auto off_time = DELAY + WIDTH;   // Non-compliant - wraps to 4464
}
```

In the following example, the sub-expression `( 0u - 1u )` results in unsigned integer wrapping in the initialization of `x`. However, in the initialization of `y`, the sub-expression is never evaluated and the expression is therefore compliant.

```
void g( bool b )
{
  uint16_t x = b ? 0u : ( 0u - 1u );               // Non-compliant
  uint16_t y = ( 0u == 0u ) ? 0u : ( 0u - 1u );    // Compliant
}
```

Wrapping within preprocessing expressions is also non-compliant:

```
#if 1u + ( 0u - 10u )     // Non-compliant as ( 0u - 10u ) wraps
#if 11u + ( 0u - 10u )    // Non-compliant as both operations wrap
#if 11u + 0u - 10u        // Compliant
```

The rule does not apply to the following example as there are no built-in arithmetic operations with constant operands.

```
constexpr auto add( const uint16_t a, const uint16_t b )
{
  return a + b;    // References to a, b are not constant expressions.
}

constexpr auto x = add( 10000u, 60000u );    // No built-in arithmetic operation
```


Licensed to: gunjanjain
22 Apr 2025

4.9 Statements

4.9.2 Expression statement [stmt.expr]

Rule 9.2.1 An *explicit type conversion* shall not be an *expression statement*

[expr.type.conv]
[stmt.expr]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule only applies to *explicit type conversions* that use *functional notation*.

Rationale

An *explicit type conversion* that uses *functional notation* is composed of a type name followed by parentheses or braces. It creates a temporary object that is discarded at the end of the statement. This notation can appear to be very similar to the declaration of a variable, except that it does not contain a variable name.

If the intent was to declare a variable for scope-based resource management (e.g. `std::lock_guard`), the destruction side effects which were expected to occur at the end of the containing block will instead occur immediately (e.g. the lock is prematurely released).

Example

In the following example, the redundant parentheses surrounding `b_mutex` violate Rule 6.0.1.

```
void f1()
{
  std::unique_lock< std::mutex >   a_mutex;    // Declaration, rule does not apply
  std::unique_lock< std::mutex > ( b_mutex );  // Declaration, rule does not apply
}

void f2()
{
  std::scoped_lock { a_mutex };                // Non-compliant
                                               //   - locks and unlocks here
  // Unprotected
}

void f3()
{
  std::scoped_lock ( a_mutex, other_mutex );   // Non-compliant
                                               //   - locks and unlocks here
  // Unprotected
}

void f4()
{
  f( std::unique_lock { a_mutex } );           // Compliant - type conversion is
                                               //   not an expression statement
}
```

See also

Rule 6.0.1

Licensed to: gunjanjain
22 Apr 2025

4.9.3 Compound statement [stmt.block]

Rule 9.3.1 The body of an *iteration-statement* or a *selection-statement* shall be a *compound-statement*

[Koenig] 24

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The body of an *iteration-statement* (`while`, `do ... while`, `for`) or a *selection-statement* (`if`, `else`, `switch`) shall be a *compound-statement*.

Rationale

It is possible for a developer to mistakenly believe that a sequence of statements forms the body of an *iteration-statement* or *selection-statement* by virtue of their indentation. The accidental inclusion of a semi-colon after the controlling expression is a particular danger, leading to a null control statement. Using a *compound-statement* clearly defines which statements actually form the body.

Additionally, it is possible that indentation may lead a developer to associate an `else` statement with the wrong `if`.

Exception

An `if` statement that is the statement to an `else` need not be contained within a *compound-statement*.

Example

The layout for the *compound-statement* and its enclosing braces are style issues which are not addressed by this document; the style used in the following examples is not mandatory.

Maintenance to the following

```
while ( data_available )
  process_data();                // Non-compliant
```

could accidentally give

```
while ( data_available )
  process_data();                // Non-compliant
  service_watchdog();
```

where `service_watchdog` should have been added to the loop body. The use of a *compound-statement* significantly reduces the chance of this happening.

The next example appears to show that `action_2` is the `else` statement to the first `if`.

```
if ( flag_1 )
  if ( flag_2 )                  // Non-compliant
    action_1();                  // Non-compliant
else
  action_2();                    // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

when the actual behaviour is

```
if ( flag_1 )
{
  if ( flag_2 )
  {
    action_1();
  }
  else
  {
    action_2();
  }
}
```

The use of *compound-statements* ensures that `if` and `else` associations are clearly defined.

The exception allows the use of `else if`, as shown below

```
if ( flag_1 )
{
  action_1();
}
else if ( flag_2 )              // Compliant by exception
{
  action_2();
}
else { }                        // Compliant - else with empty block
```

The following example shows how a spurious semi-colon could lead to an error

```
while ( flag );                 // Non-compliant
{
  flag = fn();
}

while ( !data_available ) { }   // Compliant - loop with empty body
```

4.9.4 Selection statements [stmt.select]

Rule 9.4.1 All `if ... else if` constructs shall be terminated with an `else` statement

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A final `else` shall always be provided whenever an `if` statement is followed by a sequence of one or more `else if` constructs.

Note: a final `else` statement is not required for a simple `if` statement.

Rationale

Terminating a sequence of `if ... else if` constructs with an `else` statement is defensive programming, complementing the requirement for a `default` clause in a `switch` statement (see Rule 9.4.2).

The addition of an `else` statement, even when empty, indicates that consideration has been given regarding the behaviour when all other conditions evaluate to `false`.

Licensed to: gunjanjain
22 Apr 2025

Example

```
void f1( bool flag_1, bool flag_2 )
{
  if ( flag_1 )
  {
    action_1();
  }
  else if ( flag_2 )
  {
    action_2();
  }                       // Non-compliant
}

void f2(bool flag_1, bool flag_2)
{
  if ( flag_1 )
  {
    action_1();
  }
  else if ( flag_2 )
  {
    action_2();
  }
  else                    // Compliant
  {
  }
}

void f3( bool flag )
{
  if ( flag )
  {
    action_1();
  }                       // Simple 'if' - rule does not apply
}
```

See also

Rule 9.4.2

Rule 9.4.2 The structure of a `switch` statement shall be appropriate

[stmt.switch]
[dcl.attr.fallthrough]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The substatement of a `switch` statement is called the *switch body*. It shall be a *compound statement*.

A *labeled statement*, along with the complete chain of its substatements that are also *labeled statements*, is called a *label group*. A *label group* that is *directly enclosed* by a *switch body* is called a *switch label group*.

The statements directly enclosed by a *switch body* are partitioned into *switch branches*, with each *switch label group* starting a new branch.

Licensed to: gunjanjain
22 Apr 2025

A `switch` statement is structured appropriately when it conforms to the following restrictions:

1. The *condition* shall only be preceded by an optional *simple-declaration*;
2. `case` or `default` *labeled statements* shall only appear as part of a *switch label group*;
3. *Switch label groups* shall only contain `case` or `default` *labeled statements*;
4. The first statement in a *switch body* shall be a *switch label group*;
5. Every *switch branch* shall be unconditionally terminated by either:
   a. A `break` statement; or
   b. A `continue` statement; or
   c. A `return` statement; or
   d. A `goto` statement; or
   e. A `throw` expression; or
   f. A call to a `[[noreturn]]` function; or
   g. A `[[fallthrough]]` attribute applied to a null statement.
6. Every `switch` statement shall have at least two *switch branches*;
7. Every `switch` statement shall have a `default` label, appearing as either the first label of the first *switch label group* or as the last label of the last *switch label group*.

Rationale

The syntax for the `switch` statement can be used to create complex, unstructured code. This rule places restrictions on the use of the `switch` statement in order to impose a simple and consistent structure:

1. A *simple-declaration* is permitted before the *condition* as it allows the declaration of a variable with restricted scope within a `switch` statement. An *expression-statement* is not permitted as the *init-statement*, as it introduces complexity without any extra benefit.
2. The C++ Standard permits a `case` label or `default` label to be placed before any statement contained within the body of a `switch` statement, potentially leading to unstructured code. To prevent this, a `case` label or `default` label is only permitted to appear at the outermost level of the compound statement forming the body of a `switch` statement.
3. Including labels other than `case` or `default` in a *switch label group* potentially allows unstructured control flow to be introduced.
4. A statement placed before a *switch label group* would either be an uninitialized variable or unreachable code.
5. If a developer fails to terminate a *switch branch*, then control flow "falls" into the following *switch branch* or, if there is no such branch, off the end and into the statement following the `switch` statement. The requirement for unconditional termination ensures that unintentional fall-throughs can be detected, with the `[[fallthrough]]` attribute being used to explicitly indicate when fall-through is intentional. *Note:* fall-through that occurs between two consecutive `case` or `default` labels having no intervening statements is not ambiguous, and is permitted by this rule.
6. A `switch` statement with a single *switch branch* is not permitted as it may be indicative of a programming error.
7. The requirement for a `default` label is defensive programming, complementing the requirement for `if ... else if` constructs to be terminated with an `else` (see Rule 9.4.1). The addition of a `default`, even when empty, indicates that consideration has been given regarding the behaviour when all other cases are not selected. Placing the `default` as the first or last label makes it easier to locate during code review.

Licensed to: gunjanjain
22 Apr 2025

Note: even when the *condition* of a `switch` has `enum` type, listing all enumerators values in `case` labels does not make the use of `default` redundant as the value could still lie outside of the set of enumerators.

Exception

If the *condition* of a `switch` statement is an *unscoped enumeration* type that does not have a *fixed underlying type*, and all the enumerators are listed in `case` labels, then a `default` label is not required. *Note:* compliance with Rule 10.2.3 ensures that an object of such a type cannot be assigned values outside of its set of enumerators.

Example

The following `switch` statement has four *switch branches*:

```
switch ( int8_t x = f(); x ) // Compliant - declaration of x is simple
{
  case 1:
  {
    break;                   // Compliant - branch unconditionally terminated
  }

  case 2:
  case 3:
    throw;                   // Compliant - branch unconditionally terminated

  case 4:
    a++;
    [[fallthrough]];         // Compliant - branch has explicit fall-through
  default:                   // Compliant - default is last label
    b++;
    return;                  // Compliant - branches unconditionally terminated
}
```

The following `switch` statement has four *switch branches*:

```
switch ( x = f(); x )        // Non-compliant - x = f() is not a simple-declaration
{
  int32_t i;                 // Non-compliant - not a switch label group

  case 5:
    if ( ... )
    {
      break;
    }
    else
    {
      break;
    }
                             // Non-compliant - termination is not unconditional

  case 6:
    a = b;                   // Non-compliant - non-empty, implicit fall-through
  case 7:
    {
      case 8:                // Non-compliant - case not in a switch label group
        DoIt();
    }
    break;
}                            // Non-compliant - default is required
```


Licensed to: gunjanjain
22 Apr 2025

```
switch ( x )                    // Non-compliant - only one switch branch
{
  default:
    ;                           // Non-compliant - default must also be terminated
}

enum Colours { RED, GREEN, BLUE } colour;

switch ( colour )
{
  case RED:
    break;
  case GREEN:
    break;
}                               // Non-compliant - default is required

switch ( colour )
{
  case RED:
  case GREEN:
    break;
  case BLUE:
    break;
}                               // Compliant by exception - all enumerators listed

switch ( colour )               // Non-compliant - only one switch branch
{
  case RED:
  default:                      // Non-compliant - default must be first or last label
  case BLUE:
    break;
}
```

See also

Rule 9.4.1, Rule 9.6.2, Rule 9.6.3, Rule 10.2.3

4.9.5 Iteration statements [stmt.iter]

Rule 9.5.1 *Legacy for statements* should be *simple*

[stmt.for]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

A *legacy for statement* is *simple* when:

1. The *init-statement* only declares and initializes a *loop-counter* of integer type; and
2. The *condition* only compares the *loop-counter* to a *loop-bound* using a *relational operator*; and
3. The *loop-counter* is modified, but only by incrementing or decrementing by a *loop-step* within the *expression* of the *for* statement; and
4. The *loop-bound* and *loop-counter* have the same type, or the *loop-bound* is a constant expression and the type of the *loop-counter* has a range large enough to represent the value of the *loop-bound*; and

Licensed to: gunjanjain
22 Apr 2025

5. The *loop-bound* and *loop-step* are *constant-expressions* or are variables that are not modified within the *for* statement; and
6. The *loop-counter*, *loop-bound* and *loop-step* are not bound to non-*const* references and do not have any of their addresses *assigned* to pointers to non-*const*.

Note: the *range-for* statement is not a *legacy for statement*.

Rationale

The number of iterations of a *legacy for statement* is determined by a user-provided loop condition and code review, which may be non-trivial, is required to ensure that the loop behaves as expected. This review is not required for iterator-based algorithms or *range-for* statements, as the number of iterations is not determined by a user-provided loop condition. It is therefore recommended that *legacy for statements* should not be used, unless they are *simple*.

It is generally unnecessary to use the *legacy for statements* as C++ Standard Library algorithms are provided for most iteration use-cases. Iterating over the contents of a container can be achieved by the use of a *range-for* statement when the existing algorithms are not suitable. Using or implementing a range adapter or iterator adapters allows *range-for* statements or iterator-based algorithms to be used to loop over other data sources and sinks.

When a *legacy for statement* cannot be replaced by an existing C++ Standard Library algorithm, it can be abstracted and confined within a (potentially generic) dedicated function to make code review and justification easier.

Note: care must be taken to ensure that a *simple legacy for statement* will make progress and terminate.

Example

```
for ( int32_t i = 0; i < 10; ++i )          // Compliant
{
  cout << i << " ";
}

bool foo( int32_t & );

for ( int32_t i = 0; i < 10; ++i )          // Non-compliant
{
  foo( i );                                 // i passed as non const & parameter
}

for ( uint32_t i = 0u; i < u64a; ++i )      // Non-compliant - loop-counter and
{                                           //   loop-bound have different types
  // ...
}

int32_t sum { };
std::array< int32_t, 10 > arr { };

for ( auto i = 0u; i < arr.size(); ++i )    // Compliant- arr.size() is constant
{
  sum += arr[ i ];
}
```

The following achieve the same without the use of *legacy for statements*:

```
for ( auto const e : arr )                  // Rule does not apply
{
  sum += e;
}
```


Licensed to: gunjanjain
22 Apr 2025

```
sum = reduce( begin( arr ),
              end( arr ),
              int32_t {} );                    // Rule does not apply
```

See also

Rule 0.0.2

Rule 9.5.2 A *for-range-initializer* shall contain at most one function call

[stmt.ranged]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The *for-range-initializer* occurs within the range-based `for` statement:

`for` (*for-range-declaration* `:` *for-range-initializer*) *statement*

For the purposes of this rule, the following are also considered to be function calls:

- Any expression creating an object of `class` type; and
- Any use of an overloaded operator.

Rationale

Compliance with this rule will avoid the undefined behaviour related to object lifetime violations when the *for-range-initializer* of a range-based `for` statement creates a temporary object.

The range-based `for` statement is defined within the C++ Standard as being equivalent to:

```
{
  auto && __range = for-range-initializer;
  auto    __begin = begin-expr;                // Uses __range
  auto      __end = end-expr;
  for ( ; __begin != __end; ++__begin)
  {
    for-range-declaration = *__begin;
    statement
  }
}
```

Even though lifetime extension through `__range` will extend the lifetime of the outermost temporary object of the *for-range-initializer*, it will not extend the lifetime of an intermediate temporary. The rules for temporary lifetime extension are subtle and it is easy to accidentally trigger *undefined behaviour* by accessing a temporary object after its lifetime has ended (see Rule 6.8.1).

Creating a temporary object containing a range requires a function call, and only a second call can result in creating a reference to or into it. Therefore, allowing no more than one function call eliminates the risk in a way that is decidable at the expense of prohibiting some non-problematic cases. Defining a variable holding the value of the desired *for-range-initializer* and using that variable will always be compliant with this rule.

Note: these lifetime issues with *range-for* statements have been resolved from C++23.

Licensed to: gunjanjain
22 Apr 2025

Example

```
extern std::vector < std::string > make();

void f()
{
  for ( char c: make().at( 0 ) )     // Non-compliant - two function calls
  {
  }
}

void g()
{
  auto range = make().at( 0 );       // Note that auto && would dangle

  for ( char c: range )              // Compliant - no call when using named range
  {
  }
}

void h()
{
  for ( auto s: make() )             // Compliant - single function call
  {
  }
}
```

The following shows an example that has no *undefined behaviour*, but which includes non-compliant cases as a consequence of preferring a decidable check:

```
std::vector< std::string > make( std::string_view );

void bar( std::string s )
{
  for ( auto e : make( s ) )         // Non-compliant - call to 'make' and an
  {                                  // implicit conversion to std::string_view
  }

  auto r = make( s );

  for ( auto e : r )                 // Compliant version of the above
  {
  }
```

See also

Rule 6.8.1

4.9.6 Jump statements [stmt.jump]

Rule 9.6.1 The `goto` statement should not be used

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

The use of `goto` is usually regarded as bad programming practice as it can lead to code that is difficult to understand and analyse. Restructuring code to avoid its use generally leads to code that has a lower level of complexity.

Licensed to: gunjanjain
22 Apr 2025

If this advice is not followed, Rule 9.6.2 and Rule 9.6.3 ensure that the use of `goto` does not result in code that is considered to be unstructured.

See also

Rule 9.6.2, Rule 9.6.3

Rule 9.6.2 A `goto` statement shall reference a label in a surrounding block

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A `goto` statement shall be *enclosed* in a statement that *directly encloses* its referenced label.

Rationale

The unconstrained use of `goto` can lead to programs that are extremely difficult to comprehend and analyse. However, flags may need to be introduced to give the required control flow when it is not used, with the possibility that the flags may themselves make the code less transparent than if `goto` were used. The restricted use of `goto` is therefore allowed where that use will not lead to semantics contrary to developer expectations.

This rule prohibits jumping in to nested blocks, as this results in complex control flow.

Example

```
void f1()
{
  int32_t j = 0;

  goto L1;                                  // Non-compliant

  for ( j = 0; j < 10 ; ++j )
  {
  L1:
    j;
  }
}

void f2()
{
  for ( int32_t j = 0; j < 10 ; ++j )
  {
    for ( int32_t i = 0; i < 10; ++i )
    {
      goto L2;                              // Compliant
    }
  }

L2:
  f1();
}
```


Licensed to: gunjanjain
22 Apr 2025

```
switch ( i )
{
  case 0:
    if ( x < y )
      goto L3;                          // Non-compliant
    break;

  case 1:
  L3:
    break;
}
```

See also

Rule 9.6.1, Rule 9.6.3

Rule 9.6.3 The `goto` statement shall jump to a label declared later in the function body

Category Required

Analysis Decidable, Single Translation Unit

Rationale

The unconstrained use of `goto` can lead to programs that are extremely difficult to comprehend and analyse. However, flags may need to be introduced to give the required control flow when it is not used, with the possibility that the flags may themselves make the code less transparent than if `goto` were used. The restricted use of `goto` is therefore allowed where that use will not lead to semantics contrary to developer expectations.

This rule prohibits the use of back jumps as they can be used to introduce iteration without using the well-defined iteration statements supplied by the language.

Note: the C++ Standard places restrictions on the uses of forward jumps. For example, it is not permitted to jump from a point where a local variable with initialization is not in scope to a point where it is in scope.

Example

```
void f()
{
  int32_t x = 0;

L1:
  if ( x == 10 )
  {
    goto L2;        // Compliant
  }
  else
  {
    ++x;
    goto L1;        // Non-compliant
  }

L2:
  return;
}
```

See also

Rule 9.6.1, Rule 9.6.2

Licensed to: gunjanjain
22 Apr 2025

Rule 9.6.4 A function declared with the `[[noreturn]]` attribute shall not return

[dcl.attr.noreturn]

Category Required

Analysis Undecidable, System

Amplification

Leaving a function as the result of an exception is not a return.

A function's compliance with this rule is determined independently of the context in which the function is called. For example, a Boolean parameter is treated as if it may have a value of `true` or `false`, even if all the calls expressed in the current program use a value of `true`.

Rationale

Returning from a function declared as `[[noreturn]]` results in *undefined behaviour*.

Note: a function may be declared as `[[noreturn]]` when:

1. It only exits by throwing an exception; or
2. It loops endlessly; or
3. It causes program termination.

Example

```
[[noreturn]] void kill_the_process()   // Compliant
{
  std::abort();                        // Note - std::abort is also [[noreturn]]
}

[[noreturn]] void throw_some()         // Compliant - only exits with an exception
{
  throw 42;
}

[[noreturn]] void g( bool b )          // Non-compliant - returns if 'b' is false
{
  if ( b )
  {
    throw std::exception{};
  }
}
```

See also

Rule 6.2.2

Licensed to: gunjanjain
22 Apr 2025

Rule 9.6.5 A function with non-`void` return type shall return a value on all paths

[stmt.return]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The compound statement of a lambda expression with a non-`void` return type is also considered to be a function covered by this rule.

A `return` is not required after an explicit `throw` or after calling a function marked `[[noreturn]]`.

This rule does not apply to `main`, as it implicitly returns `0` if an exit path does not explicitly return a value.

Note: flowing off the end of a function body, except within `main`, is equivalent to a `return` with no operand.

Rationale

The operand to `return` gives the value that the function returns. The absence of a `return` with an operand in an execution path through a function with a non-`void` return type results in *undefined behaviour*.

Example

```
int32_t fn1()               // Non-compliant
{
  // No return
}

int32_t fn2( int32_t x )    // Compliant
{
  if ( x > 100 )
  {
    throw 42;               // Exiting via an exception
  }
  else
  {
    return x;               // Value returned on other path
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

4.10 Declarations

4.10.0 MISRA [misra]

Rule 10.0.1 A *declaration* should not declare more than one variable or member variable

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

An *init-declarator-list* or a *member-declarator-list* should consist of a single *init-declarator* or *member-declarator* respectively.

Structured bindings are permitted by this rule.

Rationale

Where multiple declarators appear in the same *declaration*, the type of an identifier may not meet developer expectations.

Example

```
int32_t   i1; int32_t j1;                // Compliant
int32_t   i2, * j2;                      // Non-compliant
int32_t * i3,
        & j3 = i2;                       // Non-compliant

struct point
{
  int32_t x, y;                          // Non-compliant
};

std::map< char, char > map = f();

auto [ loc, inserted ] =
  map.insert( make_pair( 'A', 'a' ) );   // Compliant - structured binding
```

4.10.1 Specifiers [dcl.spec]

Rule 10.1.1 The target type of a pointer or *lvalue* reference parameter should be const-qualified appropriately

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

The target type of a named pointer or reference parameter should be const-qualified, unless:

1. It is not an *object type*; or
2. The parameter is *assigned* to a pointer or reference with a non-const target type; or
3. The target object is modified within the function.

Licensed to: gunjanjain
22 Apr 2025

For the purposes of this rule, an object is also considered to be modified if it is passed as a pointer to non-const parameter or a non-const reference parameter, including use as the implicit `this` parameter of a non-const member function.

This rule does not apply to parameters:

1. That are unnamed; or
2. Of virtual functions; or
3. Of function templates; or
4. Of functions or lambdas declared within the scope of a template.

Note: this rule also applies to pointer parameters declared using array syntax.

Rationale

Consistent application of this guideline results in function signatures that more accurately reflect the behaviour of the functions within the project, making it less likely that a developer will falsely assume that a call will not result in the modification to an object.

The rule does not apply to virtual functions as different overrides of the function may or may not modify the target object, and all overrides will need to omit const-qualification if one or more of the overrides requires that the target type be non-const. Similarly, for templates, only some instantiations may modify the target object.

Exception

This rule does not apply to `main` whose signature, which does not use const-qualification, is defined within the C++ Standard.

Example

```
void f1(       int8_t *       p1,     // Compliant - *p1 modified
         const int8_t *       p2,     // Compliant - *p2 not modified, but is const
               int8_t *       p3,     // Non-compliant - *p3 not modified, no const
               int8_t * const p4,     // Non-compliant - *p4 not modified, no const
               int8_t         a[3] )  // Non-compliant - 'a' decays to int8_t *
{
   *p1 = *p2 + *p3 + *p4 + a[ 2 ];
}

auto & f2( int32_t &  i,              // Compliant
           int32_t && j,              // Rule does not apply - rvalue reference
           int32_t &    )             // Rule does not apply - unnamed parameter
{
  return i;                           // Assigning to non-const reference
}

auto f3( std::vector< int32_t > & x )  // Compliant - even though x.begin has an
{                                      //             equivalent const overload
  return x.begin();                    // Non-const member function
}

auto f4( std::vector< int32_t > & x )  // Non-compliant
{
  return x.cbegin();                   // Const member function
}
```


Licensed to: gunjanjain
22 Apr 2025

```
template< typename T >
struct A
{
  void foo ( T       & t,              // Rule does not apply - in template scope
             int32_t & i )             // Rule does not apply - in template scope
  {
    t.f( i );                          // t and/or i may or may not be modified,
  }                                    // depending on the signature of T::f
};
```

Rule 10.1.2 The `volatile` qualifier shall be used appropriately

Category Required

Analysis Decidable, Single Translation Unit

Amplification

It is inappropriate to declare the following *entities* as `volatile`:

- Local variables;
- Function parameters;
- Function return types;
- Member functions;
- Structured bindings.

Note: a pointer or reference to a `volatile` *entity* is permitted.

Rationale

While the C++ Standard permits `volatile` qualification to be applied to the *entities* listed above, the behaviour is not well-defined or well-understood. In addition, `volatile` does not prevent data races, but it is often incorrectly used when trying to ensure thread safety.

Note: some of these uses of `volatile` have been deprecated in C++20 and their removal is planned for a future version.

Example

```
void f1( volatile int32_t i )    // Non-compliant
{
  use< int32_t >( i );
}

void f2( volatile int32_t * p )  // Compliant - parameter is not volatile
{
  use< int32_t * >( p );
}

void f3( int32_t * volatile p )  // Non-compliant - parameter is volatile
{
  use< int32_t * >( p );
}

void f4( int32_t i )
{
  volatile int32_t j = i;        // Non-compliant

  use< int32_t >( j );
}

volatile int32_t f5();           // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

```
void f6()
{
  int32_t g[ 2 ] = { 1, 2 };

  auto volatile [ a, b ] = g;     // Non-compliant
}

struct S
{
  volatile uint32_t reg;          // Compliant
};

void f7( S s );                   // Compliant - but unlikely to work as expected
void f8( S & s );                 // Compliant - preserves volatile behaviour of reg
```

4.10.2 Enumeration declarations [dcl.enum]

Rule 10.2.1 An enumeration shall be defined with an explicit underlying type

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The underlying type of an `enum` is explicit when its declaration has an *enum-base*.

Additionally, an explicit or implicit enumerator value shall not be the result of a narrowing conversion.

Note: the C++ Standard states that any program that violates this additional requirement is ill-formed. However, it is known that some compilers do not issue a diagnostic.

Rationale

When an `enum` is defined without an *enum-base*:

- If the `enum` is *unscoped*, the underlying type is *implementation-defined*, with the only restriction being that the type must be able to represent the enumeration values; or
- If the `enum` is *scoped*, it will implicitly have an underlying type of `int`.

In both cases, using an explicit underlying type ensures that this type is obvious to the user, reducing the risk of an operation on enumerators leading to unwanted integer overflows.

Exception

The underlying type does not have to be specified when:

1. All of the enumerators in an enumeration use their default values — these enumerators are typically used as symbolic values, meaning the underlying type is not important (Rule 10.2.3 restricts which operations are permitted for such types); or
2. An enumeration is declared in an `extern "C"` block — i.e. the enumeration is intended to be used with C code.

Licensed to: gunjanjain
22 Apr 2025

Example

```
enum class Enum1 : int8_t     // Compliant
{
  E0 = 1,
  E1 = 2,
  E2 = 4
};

enum class Enum2              // Non-compliant - no explicit underlying type
{
  E0 = 0,
  E1,
  E2
};
```

The following example will be reported as ill-formed by a conforming compiler.

```
enum class Enum3 : uint8_t    // Non-compliant - cannot represent value for E2
{                             // Implicit value is the result of wrapping
  E0,
  E1 = 255,
  E2
};

enum class Enum4              // Compliant by exception #1
{
  E0,
  E1,
  E2
};

extern "C"
{
  enum Enum5                  // Compliant by exception #2
  {
    E7_0 = 0,
    E7_1,
    E7_2
  };
}
```

See also

Rule 10.2.3

Rule 10.2.2 *Unscoped enumerations* should not be declared

[dcl.enum] / 2

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

If an *unscoped enumeration* type is declared, its enumerators may hide an *entity* declared with the same name in an outer scope. This may lead to developer confusion.

Using a *scoped enumeration* restricts the scope of its enumerators' names, which can only be referenced as qualified names. In addition, its enumerators cannot be implicitly converted to numeric values.

Licensed to: gunjanjain
22 Apr 2025

Exception

This rule does not apply to an *unscoped enumeration* type declared as a class member as any name hiding would be reported as a violation of Rule 6.4.1. This idiom was commonly used before *scoped enumeration* types were introduced.

Example

```
static int32_t E10 = 5;
static int32_t E20 = 5;

enum       E1 : int32_t { E10, E11, E12 };    // Non-compliant - ill-formed as
                                              // E10 already declared
enum class E2 : int32_t { E20, E21, E22 };    // Compliant

void f1( int32_t number );

void f2()
{
  f1( 0 );
  f1( E11 );        // Implicit conversion from enum to int32_t type
  f1( E2::E21 );    // Ill-formed - implicit conversion of scoped enumeration

  f1( static_cast< int32_t >( E2::E21 ) );    // Explicit conversion needed
}

class C1
{
public:
  enum Cstyle { E10, E20, E30 };              // Compliant by exception
};
```

See also

Rule 6.4.1

Rule 10.2.3 The numeric value of an *unscoped enumeration* with no fixed *underlying type* shall not be used

[dcl.enum] Implementation 7

Category Required

Analysis Decidable, Single Translation Unit

Amplification

In an evaluated context, expressions of *unscoped enumeration* type without a fixed *underlying type* shall not be used:

- As operands to an *arithmetic*, *bitwise*, *shift*, *logical*, or *compound assignment* operator;
- As operands to *relational* and *equality* operators, unless both operands have the same enumeration type;
- As the source of an *assignment* or a `static_cast`, unless the target has the same enumeration type or is an integer type large enough to accept all the values of the narrowest possible underlying type;
- As the *condition* of a `switch`, unless all `case` constants are enumerators of the same enumeration.

Licensed to: gunjanjain
22 Apr 2025

Additionally, a `static_cast` expression shall only have an *unscoped enumeration* target type if that enumeration type has a fixed *underlying type*.

Rationale

The *underlying type* of an *unscoped enumeration* that does not have a fixed *underlying type* is *implementation-defined*, so any implicit conversion could yield surprising results.

Example

```
enum E     { e1a, e1b };
enum Other { e2a };

void g( int32_t i );

void f( E e )
{
  E       e2 = e;       // Compliant - assignment to the same type
  int32_t i1 = e;       // Compliant - assignment to a large enough integer

  e == e1a;             // Compliant
  e < e1b;              // Compliant
  e == e2a;             // Non-compliant - second operand of a different type
  e + 1;                // Non-compliant - addition

  g( e );               // Compliant - assignment to large enough integer

  switch( e )           // Non-compliant - cases are not all enumerators of E
  {
    case e1b: return;   // e1b is an enumerator of E
    case e2a: return;   // e2a is not an enumerator of E
  }

  auto s  = sizeof( e + 1 );         // Unevaluated context - rule does not apply
  E    e3 = static_cast< E >( 0 );   // Non-compliant

  auto a1 = "QWERTY";
  a1[ e1a ];            // Compliant - index operator
  *( a1 + e1a );        // Non-compliant

  std::string a2 { a1 };
  a2[ e1a ];            // Compliant - assignment to a large enough integer (size_t)
}
```

See also

Rule 10.2.2

4.10.3 Namespaces [basic.namespace]

Rule 10.3.1 There should be no unnamed namespaces in *header files*

[namespace.unnamed]

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

An unnamed namespace is unique within each *translation unit*. Any *declarations* appearing in an unnamed namespace within a *header file* refer to different *entities* in each *translation unit*, which might not be consistent with developer expectations.

Licensed to: gunjanjain
22 Apr 2025

Example

```
// Header.hpp
namespace            // Non-compliant
{
  inline int32_t x;
}

void fn_a();

// File1.cpp
#include "Header.hpp"

void fn_a()
{
  x = 42;
}

// File2.cpp
#include "Header.hpp"

void fn_b()
{
  fn_a();            // Assigns 42 to 'x' in translation unit for 'File1.cpp'

  if ( x == 42 ) {}  // 'x' within this translation unit will not have the value 42
}
```

4.10.4 The *asm* declaration [dcl.asm]

Rule 10.4.1 The `asm` declaration shall not be used

[dcl.asm] Implementation 1

Category Required

Analysis Decidable, Single Translation Unit

Rationale

The `asm` declaration is *conditionally-supported*, with the use of any assembly language insert resulting in *implementation-defined behaviour*.

Many modern development environments provide better means (such as intrinsic functions) for achieving what has traditionally been done by the use of assembly language.

Encapsulation of assembly language should be considered if this rule is subject to deviation, as this aids portability.

Note: the use of any assembly language that does not use the `asm` declaration is a language extension, and is restricted by Rule 4.1.1.

Licensed to: gunjanjain
22 Apr 2025

4.11 Declarators

4.11.3 Meaning of declarators [dcl.meaning]

Rule 11.3.1 Variables of array type should not be declared

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

A variable of array type does not have value semantics and its size has to be managed separately. It is possible to use types that do not have these limitations. For example:

- `std::array` — provides value semantics and manages the size;
- `std::string_view` — manages the size.

Exception

The declaration of an array of *const* character type is permitted when it is immediately initialized with a string literal.

Example

```
void foo() noexcept
{
  const size_t                      size { 10 };
        wchar_t                     a1 [ size ];  // Non-compliant
        std::array< wchar_t, size > a2;           // Compliant
}

void bar( int      a[ 10 ],    // Rule does not apply - pointer to int
          int ( &b )[ 10 ],    // Rule does not apply - reference to array
          int ( *c )[ 10 ])    // Rule does not apply - pointer to array
{
}

struct S
{
  std::uint16_t a3[ 10 ];                         // Non-compliant
};

using namespace std::literals;

const char s1[] = "abcd";                         // Compliant by exception
      char s2[] = "abcd";                         // Non-compliant
const auto best = "abcd"sv;                       // Compliant
```

See also

Rule 7.11.2

Licensed to: gunjanjain
22 Apr 2025

Rule 11.3.2 The *declaration* of an object should contain no more than two levels of pointer indirection

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

Any *typedef-name* appearing in a *declaration* is treated as if it were replaced by the type that it denotes.

Note: the pointer decay that occurs when declaring a function parameter of array type introduces a level of pointer indirection.

Rationale

Use of more than two levels of indirection can seriously impair the ability to understand the behaviour of the code, and therefore should be avoided.

Example

```
typedef int8_t * INTPTR1;
using INTPTR2 = int8_t *;

struct s
{
  int8_t *   s1;                                  // Compliant
  int8_t **  s2;                                  // Compliant
  int8_t *** s3;                                  // Non-compliant
};

struct s *   ps1;                                 // Compliant
struct s **  ps2;                                 // Compliant
struct s *** ps3;                                 // Non-compliant

int8_t **  (   *pfunc1 )();                       // Compliant
int8_t **  (  **pfunc2 )();                       // Compliant
int8_t **  ( ***pfunc3 )();                       // Non-compliant
int8_t *** (  **pfunc4 )();                       // Non-compliant

void function( int8_t  *   par1,                  // Compliant
               int8_t  **  par2,                  // Compliant
               int8_t  *** par3,                  // Non-compliant
               INTPTR1 *   par4,                  // Compliant
               INTPTR1 *   const * const par5,    // Non-compliant
               int8_t  *   par6[],                // Compliant
               int8_t  **  par7[],                // Non-compliant
               int8_t  ** &par8)                  // Compliant

{
   int8_t  *   ptr1;                              // Compliant
   int8_t  **  ptr2;                              // Compliant
   int8_t  *** ptr3;                              // Non-compliant
   INTPTR2 *   ptr4;                              // Compliant
   INTPTR2 *   const * const ptr5;                // Non-compliant
   int8_t  *   ptr6[ 10 ];                        // Compliant
   int8_t  **  ptr7[ 10 ];                        // Compliant
}
```

Explanation of types:

- `par1` and `ptr1` are of type pointer to `int8_t`.
- `par2` and `ptr2` are of type pointer to pointer to `int8_t`.

Licensed to: gunjanjain
22 Apr 2025

- `par3` and `ptr3` are of type pointer to a pointer to a pointer to `int8_t`. This is three levels and is non-compliant.
- `par4` and `ptr4` are expanded to a type of pointer to a pointer to `int8_t`.
- `par5` and `ptr5` are expanded to a type of const pointer to a const pointer to a pointer to `int8_t`. This is three levels and is non-compliant.
- `par6` is of type pointer to pointer to `int8_t` because arrays are converted to a pointer to the initial element of the array.
- `ptr6` is of type array of pointers to `int8_t`.
- `par7` is of type pointer to pointer to pointer to `int8_t` because arrays are converted to a pointer to the initial element of the array. This is three levels and is non-compliant.
- `ptr7` is of type array of pointer to pointer to `int8_t`. This is compliant.
- `par8` is of type reference to pointer to pointer to `int8_t`. This is compliant.

4.11.6 Initializers [dcl.init]

Rule 11.6.1 All variables should be initialized

[dcl.init]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

All variables should either be explicitly or implicitly initialized.

Apart from the following, all variables should be explicitly initialized with an associated initializer in their definition:

1. Variables of `class` type, or
2. Function parameters (which are initialized with the corresponding argument value), or
3. Variables with static storage duration (which are *zero-initialized* by default).

Rationale

Having several states within a program increases the risk of defects being introduced. Each variable that is first uninitialized, then set to a value creates two program states. It is therefore better to initialize the variable directly to a value that is to be used. The intent of this rule is not that each variable is initialized with some value, but that it is initialized with its real value; the one that will be used when the variable is next read.

In order to achieve this, the variable definition can be delayed until the "right" value is available. This naturally leads to reducing the variable's scope, reducing the risk of the variable being used inappropriately. An immediately evaluated lambda can be used to compute a value when a variable's initialization is more complex.

In many cases, initializing the variable within its definition allows it to be a constant definition.

Note: there are many ways to explicitly initialize a variable. When possible, the list-initialization syntax (with curly braces) should be used as it does not suffer from the issues that arise from the use of other syntactic forms (e.g. narrowing or declaring a function while trying to define a variable, also known as "the most vexing parse").

Licensed to: gunjanjain
22 Apr 2025

Example

```
void f( bool cond )
{
  int32_t i;                        // Non-compliant

  if ( cond ) { i =  42; }
  else        { i = -1;  }

  int32_t j = cond ? 42 : -1;       // Compliant
  int32_t k = [&]()                 // Compliant
    {
      if ( cond ) { return 42; }
      else        { return -1; }
    }();

  string s;                         // Compliant - default-initialized
}

int32_t g;                          // Compliant - static initialization applies

void f()
{
  thread_local int32_t i;           // Compliant - static initialization applies
}
```

See also

Rule 15.1.4

Rule 11.6.2 The value of an object must not be read before it has been set

[dcl.init] Undefined 12

Category Mandatory

Analysis Undecidable, System

Amplification

For the purposes of this rule, an array element or class member is considered to be a discrete object.

Note: struct members are also class members.

Rationale

Objects created with *automatic storage duration* or *dynamic storage duration* have an *indeterminate value*. Reading an indeterminate value may result in *undefined behaviour*.

This rule requires that all objects are written, either by implicit or explicit initialization in their declaration or by assignment, before they are read.

Note: jumping over an initializer by the use of a `goto` or `switch` statement "bypasses" the declaration of the object, rendering the program *ill-formed*.

Example

```
namespace
{
  int32_t Z;                    // Compliant - implicitly initialized with '0'
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void f()
{
  int32_t  i;
  int32_t  j = i + 1;         // Non-compliant - i has not been assigned a value

  int32_t * p = new int32_t;
  int32_t   k = *p;           // Non-compliant - *p has not been assigned a value

  int32_t * q;

  if ( q == p )               // Non-compliant - q has not been assigned a value
  {
  }
}

int32_t g( bool b )
{
  if ( b )
  {
    goto L1;
  }

  int32_t x;

  x = 10u;

L1:
  x = x + 1u;               // Non-compliant - x may not have been assigned a value

  return x;
}

struct S { int32_t a; int32_t b; };

void h()
{
  S s1;
  S s2 = { 10 };

  auto i1 = s1.a;           // Non-compliant
  auto i2 = s2.b;           // Compliant - s2.b implicitly initialized to 0

  int32_t array1[ 10 ] = { 1, 2, 3 };
  int32_t array2[ 10 ];
  auto i3 = array1[ 5 ];    // Compliant - array1[ 5 ] implicitly initialized to 0
  auto i4 = array2[ 5 ];    // Non-compliant
}

class C
{
public:
  C() : m_a( 10 ), m_b( 7 )   // Both m_a and m_b initialized
  {
  }

  C( int32_t a ) : m_a( a )   // m_b not initialized
  {
  }

  int32_t GetmB()
  {
    return m_b;
  }
```


Licensed to: gunjanjain
22 Apr 2025

```
private:
  int32_t m_a;
  int32_t m_b;
};

int main()
{
  C c1;
  if ( c1.GetmB() > 0 )       // Compliant - m_b initialized
  {
  }

  C c2( 5 );

  if ( c2.GetmB() > 0 )       // Non-compliant - m_b not initialized
  {
  }
}
```

See also

Rule 15.1.4

Rule 11.6.3 Within an enumerator list, the value of an implicitly-specified *enumeration constant* shall be unique

Category Required

Analysis Decidable, Single Translation Unit

Rationale

An implicitly-specified *enumeration constant* has a value one greater than its predecessor. If the first *enumeration constant* is implicitly-specified, then its value is zero.

An explicitly-specified *enumeration constant* has the value of the associated constant expression.

If implicitly-specified and explicitly-specified constants are mixed within an enumeration list, it is possible for values to be duplicated. Such duplication may be unintentional and may give rise to unexpected behaviour.

This rule requires that any duplication of *enumeration constants* be made explicit, thus making the intent clear.

Exception

An implicitly-specified *enumeration constant* may have the same value as an explicitly-specified *enumeration constant* from the same enumeration when the explicitly-specified value is defined using only the name of another *enumeration constant*.

Example

The following examples are compliant as it is clear which *enumeration constants* have the same value:

```
enum E1 { A = 3, B, C = 5, D = 5 };          // Compliant
enum E2 { A = 3, B, C,     D = C, E = D };   // Compliant by exception
```

The following examples are non-compliant as *enumeration constants* have the same implicit values:

```
enum E3 { A = 3, B, C, D = 4    };           // 'B' and 'D' have the same value
enum E4 { A = 3, B, C, D = B, E };           // 'C' and 'E' have the same value
```


Licensed to: gunjanjain
22 Apr 2025

The following example is non-compliant as the use of `B + 1` means the exception does not apply.

```
enum E5 { A = 3, B, C, D = B + 1 };            // Non-compliant
```

4.12 Classes

4.12.2 Class members [class.mem]

Rule 12.2.1 Bit-fields should not be declared

[class.bit] Implementation 1

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

There are a number of aspects of bit-fields that a developer needs to consider, including:

- It is *implementation-defined* whether the bit-fields are allocated from the high or low end of a storage unit (usually a byte);
- It is *implementation-defined* whether or not a bit-field can overlap a storage unit boundary (e.g. if a 6-bit bit-field and a 4-bit bit-field are declared in that order, then the 4-bit bit-field may either start a new byte or it may use 2 bits in one byte and 2 bits in the next);
- If the bit-field's width is greater than the number of bits in the object representation of the bit-field's type, then the extra bits are padding bits and do not participate in the value representation of the bit-field.

These issues are generally benign (e.g. when packing together short-length data to save storage space), but they may lead to errors if the absolute position of the bit-fields is important (e.g. when accessing hardware registers).

Provided the elements of the structure are only accessed by name, the developer need make no assumptions about the way that the bit-fields are stored within the structure.

Example

```
struct message
{
  unsigned char low  : 4;    // Non-compliant
  unsigned char high : 4;    // Non-compliant
};
```


Licensed to: gunjanjain
22 Apr 2025

Rule 12.2.2 A bit-field shall have an appropriate type

[class.bit] / 3, 4; Implementation 1
[basic.fundamental] / 5; Implementation 1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The following types are appropriate for a bit-field:

- *Signed* and *unsigned integer types*;
- An `enum` with a fixed underlying type of *signed* or *unsigned integer type*, provided that all of its enumeration values are representable within the width of the bit-field;
- `bool`.

Rationale

The `char` and `wchar_t` types shall not be used for bit-fields as it is *implementation-defined* if they are signed or unsigned. The `char16_t` and `char32_t` types are not permitted as they are only intended to be used to store character code points.

Similarly, when using an unscoped `enum` without specifying the underlying type, it is *implementation-defined* if the underlying representation is signed or unsigned. Therefore, the exact number of bits required to represent all values in the enumeration is also *implementation-defined*.

Example

```
struct S
{
   signed int a : 2;    // Compliant
          int   : 2;    // Compliant
      int32_t b : 2;    // Compliant
         char c : 2;    // Non-compliant

  signed char d : 2;    // Compliant - signed integer type
      wchar_t e : 2;    // Non-compliant - not a signed or unsigned integer type
     char32_t f : 2;    // Non-compliant - not a signed or unsigned integer type
         bool g : 1;    // Compliant
};

enum Direction { Top, Left, Bottom, Right };
enum Colour : char          { Red, Pink, Blue };
enum Line   : unsigned char { Plain, Dash, Dot};

struct S
{
  Direction dir        : 4;   // Non-compliant - unscoped and no underlying type
  Colour    lineColour : 2;   // Non-compliant - underlying type is plain char
  Line      lineStyle1 : 1;   // Non-compliant - cannot represent Dot
  Line      lineStyle2 : 2;   // Compliant
};
```


Licensed to: gunjanjain
22 Apr 2025

Rule 12.2.3 A named bit-field with *signed integer type* shall not have a length of one bit

Category Required

Analysis Decidable, Single Translation Unit

Rationale

A single-bit signed bit-field is unlikely to behave in a useful way and its presence is likely to indicate an error.

Note: anonymous signed bit-fields of any length are allowed as they cannot be accessed.

Example

```
struct S
{
  signed int a : 1;    // Non-compliant
  signed int   : 1;    // Rule does not apply
  signed int   : 0;    // Rule does not apply
  signed int b : 2;    // Compliant
         int c : 1;    // Non-compliant
};
```

4.12.3 Unions [class.union]

Rule 12.3.1 The `union` keyword shall not be used

[class.union] / 1, 5
[basic.life] 1; Undefined 4

Category Required

Analysis Decidable, Single Translation Unit

Rationale

A member of a union can be written and the same member can then be read back in a well-defined manner.

However, writing to one union member and then reading back from a different union member results in *undefined behaviour*. In addition, the use of a member of non-trivial type requires manual control of its lifetime. For these reasons, unions shall not be used.

The class `std::variant`, available since C++17, provides a type-safe union that can be used to store a value of one type from a fixed set of alternatives. In contrast to unions, the alternatives are accessed by type (if the types are different) or index, not by name. It is impossible to access an inactive member of a `std::variant`. For example, trying to access an inactive member via `std::get` will lead to an exception being thrown.

Example

```
union Data1                                    // Non-compliant
{
  int32_t i;
  float   j;
};

using Data2 = std::variant< int32_t, float >;  // Rule does not apply
```


Licensed to: gunjanjain
22 Apr 2025

4.13 Derived classes

4.13.1 Multiple base classes [class.mi]

Rule 13.1.1 Classes should not be inherited virtually

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

Virtual inheritance of base classes is not recommended as it introduces a number of potentially confusing behaviours, such as call by dominance in diamond hierarchies and changes to the order of initialization of bases.

Example

```
struct A
{
  virtual int32_t foo() { return 1; }
};

struct B : public virtual A            // Non-compliant
{
  int32_t goo()
  {
    return foo();
  }
};

struct C : public virtual A            // Non-compliant
{
  int32_t foo() override { return 2; }
};

struct D : C, B
{
};

int main()
{
  D d;

  return d.goo();                      // Calls C::foo(), which may not be expected
}
```

See also

Rule 8.2.1, Rule 13.1.2, Rule 15.1.1

Licensed to: gunjanjain
22 Apr 2025

Rule 13.1.2 An accessible base class shall not be both virtual and non-virtual in the same hierarchy

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Where a base class is inherited both virtually and non-virtually, it is unclear if the intent is for there to be one or more instances of the base class subobject.

Example

```
class A {};

class B1: public virtual A {};
class B2: public virtual A {};
class B3: public         A {};

class C:  public B1, B2, B3 {};  // Non-compliant - C has two A subobjects
```

See also

Rule 13.1.1

4.13.3 Virtual functions [class.virtual]

Rule 13.3.1 *User-declared* member functions shall use the `virtual`, `override` and `final` specifiers appropriately

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The specifiers are used appropriately when a member function declaration:

1. Does not override a function in a base class, and has either no specifier or has the `virtual` specifier; or
2. Overrides a function in a base class, does not use the `virtual` specifier, and does use either the `override` or `final` specifier.

Note: this rule also applies to destructors.

Rationale

When a function is declared that does not override a function in a base class (including the case where the owning class has no base classes), then it is either not intended to be virtual or it is a virtual function that is expected to be overridden in a derived class. The function declaration should therefore include either no specifier or the `virtual` specifier, as appropriate. The use of the `override` specifier in this case would render the program *ill-formed*, whilst use of the `final` specifier would mean that it is a virtual function that cannot be subsequently overridden (in which case making it `virtual` is redundant).

Licensed to: gunjanjain
22 Apr 2025

When a function is declared that overrides a virtual function in a base class:

- The `override` specifier explicitly documents that this declaration overrides a function in a base class;
- The `final` specifier documents that no further overrides are permitted.

Whilst they are permitted by the C++ Standard, the following redundant combinations of specifier shall be avoided:

1. Use of `virtual` with either `override` or `final`;
2. Use of `final` with `override`.

The use of a single specifier makes the meaning clearer:

1. `virtual` — this is a new virtual function this is expected to be overridden;
2. `override` — this is an override that may or may not be overridden;
3. `final` — this is an override that cannot be overridden.

Notes:

1. Declaring a class itself as `final` does not make its virtual member functions `override` or `final`; the compiler is not required to check that the declarations are overrides.
2. Rule 6.4.2 restricts the use of function declarations that hide non-virtual functions in base classes.

Example

```
class A
{
public:
  virtual ~A() = default;
  virtual void f1() noexcept = 0;                // Compliant
  virtual void f2() noexcept {}                  // Compliant
  virtual void f3() noexcept {}                  // Compliant
          void f4() noexcept {}                  // Compliant

  // The following declarations are non-compliant
  virtual void f5() noexcept final = 0;          // 'virtual' and 'final'
  virtual void f6() noexcept final {}            // 'virtual' and 'final'
          void f7() noexcept final {}            // Ill-formed - not virtual
};

class B : public A
{
public:
  // The following declarations are non-compliant
  ~B();                                          // No specifier given for override
  virtual void f1() noexcept override {}         // 'virtual' and 'override'
          void f2() noexcept override final {}   // 'override' and 'final'
          void f3() noexcept {}                  // No specifier given for override
          void f4() noexcept override {}         // Ill-formed - A::f4() not virtual
};
```

See also

Rule 6.4.2, Rule 15.0.1

Licensed to: gunjanjain
22 Apr 2025

Rule 13.3.2 Parameters in an overriding virtual function shall not specify different default arguments

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Each parameter in an overriding virtual function shall either:

1. Not specify a default argument; or
2. Use a *constant expression* as its default argument, with the corresponding parameter in the non-overriding function also specifying a default argument that is a *constant expression* with the same value.

Rationale

Default arguments are determined by the static type of the object. If a default argument is different for a parameter in an overriding function, the value used in the call will be different when calls are made via the base or derived object, which may be contrary to developer expectations.

Requiring that multiple default arguments for the same parameter be *constant expressions* allows compliance checks for this rule to be decidable.

Example

```
int32_t x();

class Base
{
public:
  virtual void good1( int32_t a = 0   );
  virtual void good2( int32_t a = x() );
  virtual void bad1 ( int32_t a = 0   );
  virtual void bad2 ( int32_t a       );
  virtual void bad3 ( int32_t a = x() );
};

class Derived : public Base
{
public:
  void good1( int32_t a = 0   ) override;   // Compliant - same default used
  void good2( int32_t a       ) override;   // Compliant - no default specified
  void bad1 ( int32_t a = 1   ) override;   // Non-compliant - different value
  void bad2 ( int32_t a = 2   ) override;   // Non-compliant - no default in base
  void bad3 ( int32_t a = x() ) override;   // Non-compliant - not constant
};

void f( Derived & d )
{
  Base & b = d;

  b.good1();       // Will use default of 0
  d.good1();       // Will use default of 0
  b.good2();       // Will use default of x( )
  d.good2( 0 );    // No default value available to use

  b.bad1();        // Will use default of 0
  d.bad1();        // Will use default of 1
  b.bad2( 0 );     // No default value available to use
  d.bad2();        // Will use default of 2
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 13.3.3 The parameters in all *declarations* or overrides of a function shall either be unnamed or have identical names

Category Required

Analysis Decidable, System

Rationale

The name given to a parameter helps document the purpose of the parameter. If a function parameter is renamed in a subsequent *declaration*, then having different names for the same object may lead to developer confusion.

Example

The following example is compliant:

```
void fn1( int32_t a );
void fn1( int32_t   );
```

The following example is non-compliant as the parameter names have been swapped:

```
void CreateRectangle( uint32_t Height, uint32_t Width  );
void CreateRectangle( uint32_t Width,  uint32_t Height );
```

The following example is non-compliant as the named parameters are different:

```
void fn2( int32_t a );
void fn2( int32_t b ) { }
```

The following example is non-compliant as the parameter name in the override differs from the parameter name in the overridden function:

```
class Shape
{
  virtual void draw( Canvas & destination ) = 0;
};

class Rectangle : public Shape
{
  void draw( Canvas & canvas ) override;
};
```

The rule does not apply to the following example as the specialization is a different *declaration* (note that this example is non-compliant with Rule 17.8.1):

```
template< class T > void f( T t );
template<> void f< int32_t >( int32_t i );
```


Licensed to: gunjanjain
22 Apr 2025

Rule 13.3.4 A comparison of a *potentially virtual* pointer to member function shall only be with `nullptr`

[expr.eq] Unspecified 3

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A pointer to member function is *potentially virtual* if it is:

1. A compile-time constant that points to a virtual member function; or
2. A pointer to member function of a class that is incomplete at the end of the translation unit; or
3. Not a compile-time constant pointer to member function and has a type matching that of a virtual member function of its class.

Rationale

The result of comparing a pointer to member function that points to a virtual function with anything other than `nullptr` is unspecified.

Example

```
class A
{
public:
          void f1();
          void f2();
  virtual void f3();
};

void foo()
{
  if ( &A::f1 != &A::f2   ) {}   // Compliant
  if ( &A::f1 != nullptr ) {}   // Compliant
  if ( &A::f3 == &A::f2   ) {}   // Non-compliant - f3 virtual
  if ( &A::f3 == nullptr ) {}   // Compliant
}

void bar( void ( A::*ptr )() )
{
  if ( ptr == &A::f2 ) {}       // Non-compliant - ptr potentially points to A::f3,
                                // which is virtual
}
```

Note: the example above would be compliant if A had no virtual members.

```
class B
{
public:
          void f1();
          void f2();
  virtual void f3( int32_t i );
};

void bar( void ( B::*ptr )() )
{
  if ( ptr == &B::f2 ) {}       // Compliant - there are no virtual functions
                                // in B with the appropriate signature
}
```


Licensed to: gunjanjain
22 Apr 2025

```
class D: public A                // Inherits virtual functions from A
{
public:
  void f4();
};

void car( void ( D::*ptr )() )
{
  if ( ptr == &D::f4 ) {}        // Non-compliant - ptr potentially points to A::f3,
                                 // which is virtual
}

struct E;

void foo ( void ( E::*p1 )(), void ( E::*p2 )() )
{
  if ( p1 == p2 ) {}             // Non-compliant - 'E' is incomplete, so it is
                                 // unknown if the pointers are to virtual members

}

// The following definition of E anywhere in the translation
// unit would make the above example compliant
// struct E{ void f1(); void f2(); };
```

4.14 Member access control

4.14.1 Access specifiers [class.access.spec]

Rule 14.1.1 Non-static data members should be either all `private` or all `public`

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

By implementing a class interface with member functions only and making all the class data members inaccessible, it is possible to retain more control over how the object's state can be modified. For example, enforcing an invariant for the class, or making sure that the address of a data member of a class can not be accessed by its users, making detection of possibly dangling addresses more robust.

However, some classes merely need to group together some data members without defining any invariants. For such classes, making the data members `public` simplifies the code (less code to maintain, easy use of structured bindings), therefore reducing the risk of errors.

These two situations are usually exclusive, with it being difficult to reason about a class that has both `public` and `private` data members.

The use of `protected` data members would mean that:

- The members should not be directly accessed; and
- The members can be directly accessed by any derived class, possibly breaking the invariants established by the base class.

If derived classes require privileged access to data members, those members should be `private` and `protected` functions should be defined to allow them to be manipulated.

Licensed to: gunjanjain
22 Apr 2025

Example

```
class C1                        // Non-compliant - has public and private members
{
public:
  int32_t a;

private:
  int32_t b;
};

struct C2                       // Compliant
{
  C2( int32_t a, int32_t b ) : a{ a }, b{ b } {}
  int32_t a;
  int32_t b;
};

class C3                        // Compliant - rule does not apply to static members
{
public:
  C3( int32_t a, int32_t b ) : a{ a }, b{ b } {}
  static int32_t s;

private:
  int32_t a;
  int32_t b;
};
```

4.15 Special member functions

4.15.0 MISRA [misra]

Rule 15.0.1 *Special member functions* shall be provided appropriately

Category Required

Analysis Decidable, Single Translation Unit

Amplification

For the purposes of this rule, a *special member function* is said to be *customized* if it is *user-provided* and not *defaulted*.

All out-of-class definitions of the destructor, copy operations, and move operations for a class shall be placed in a single file.

In addition, a class shall satisfy all of the requirements defined within this rule that apply to it. For instance, a class that is used as a base class and that has a *customized* destructor shall comply with the requirements within all of the following requirement sections.

Requirements for all classes on copyability and movability

A `class T` is *copy-constructible* if the expression `std::is_copy_constructible_v< T >` is `true`, and similarly for *move-constructible*, *move-assignable*, and *copy-assignable*.

Licensed to: gunjanjain
22 Apr 2025

A class shall belong to exactly one of the following categories (other combinations are not permitted):

1. *Unmovable* — it is not *copy-constructible*, not *move-constructible*, not *copy-assignable* and not *move-assignable*; or
2. *Move-only* — it is *move-constructible* (and optionally *move-assignable*) but neither *copy-constructible* nor *copy-assignable*; or
3. *Copy-enabled* — it is *copy-constructible* and *move-constructible* and can optionally also be both *copy-assignable* and *move-assignable*.

Category	Move constructible	Move assignable	Copy constructible	Copy assignable
Unmovable	No	No	No	No
Move-only	Yes	No	No	No
	Yes	Yes	No	No
Copy-enabled	Yes	No	Yes	No
	Yes	Yes	Yes	Yes

Requirements in the presence of *customized* copy or move operations

If a class has *customized* copy or move operations, it shall have a *customized destructor*.

Requirements in the presence of *customized destructors*

The definition of any *customized destructor* shall contain at least one statement that is neither a *compound statement* nor a *null statement*. A class with such a *destructor* is regarded by this rule as managing a resource.

Additionally:

1. If the class is *unmovable*, it is defined to be a *scoped manager*.
2. If the class is *move-only*, it shall have a *customized move constructor*. If it is *move-assignable*, it shall also have a *customized move assignment operator*. Such a class is defined to be a *unique manager*.
3. If the class is *copy-enabled*, it shall have a *customized copy constructor* and its *move constructor* shall either be *customized* or not declared. If it is *copy-assignable*, it shall also have a *customized copy-assignment operator* and the move operations shall either both be *customized* or both not be declared. Such a class is defined to be a *general manager*.

Requirements in the presence of inheritance

A class that is used as a *public base class* shall either:

1. Be an *unmovable* class that has a (possibly inherited) `public virtual` *destructor*; or
2. Have a `protected` non-`virtual` *destructor*.

Note: these destructors shall either be defined as `= default` or *customized* (and non-empty).

Rationale

Language rules determining which user-declared *special member functions* suppress which of the compiler-provided functions are subtle and, for legacy reasons, the combinations produced may be semantically unsound (see [depr.impldec] in the C++ Standard).

This rule takes advantage of reasonable language-provided defaults and, in particular, avoids the need to explicitly implement these defaults within the code; code that is, or attempts to be, equivalent to the compiler-provided functions is superfluous and may have subtle behavioural differences.

More specifically, application of the "Rule of Zero" in class definitions is encouraged — i.e. when a class does not declare any of the *move constructor*, *copy constructor*, *move-assignment operator*, *copy-*

Licensed to: gunjanjain
22 Apr 2025

assignment operator, or *destructor special member functions*. A class following the "Rule of Zero" can be *unmovable*, *move-only* or *copy-enabled*, depending on the properties of the class's members and base classes.

The requirements on copyability and movability enforce the use of semantically sound combinations of the *special member functions*. In particular, they ensure that *copy-constructible* types are always *move-constructible*.

The requirements on classes with *customized destructors* cover the cases where resource management is involved and ensure that, when a class directly handles a resource, customized code will be called when an instance of the class is copied, moved or destroyed.

The requirements on base classes reduce the risk of slicing and deleting a derived class instance through a base class pointer, when the base class does not have a virtual destructor:

- Compliance with requirement 1 ensures that these risks are completely prevented.
- Compliance with requirement 2 ensures that these risks are prevented for code that does not have privileged access to the base class.

Compliance with this rule addresses the vulnerabilities covered by the "Rule of Zero", the "Rule of Five" and similar rules in other C++ guidelines. It also covers the vulnerabilities identified within the pre-C++11 "Rule of Three" — see the requirements for *general manager* (above).

Exception

An *aggregate*, which cannot have a user-declared destructor, may be used as a *public base class* — this allows empty base class optimization for mix-in and tag types.

Example

```
struct MyTagClass {};        // Compliant - Rule of Zero (empty class)
struct MyValue               // Compliant - Rule of Zero
{
  int32_t val { 42 };
};

struct PolyBaseWrong         // Non-compliant - base class that is not an aggregate
{                            //                 and has no virtual destructor.
  virtual void doIt();       //                 Additionally, slicing may occur.
};

struct DerivedWrong : PolyBaseWrong {};

struct PolyBase              // Compliant - unmovable base class with virtual public
{                            //             destructor
  virtual void doIt() = 0;
  virtual ~PolyBase() = default;

  PolyBase & operator=( PolyBase && ) = delete;   // This makes the class unmovable
};

struct Derived : PolyBase
{
  // Class definition
};
```


Licensed to: gunjanjain
22 Apr 2025

```
struct NonEmptyDestructor   // Non-compliant - copy-enabled class with a customized
                            //                 destructor but non-customized
{                           //                 copy-operations
  ~NonEmptyDestructor()     // Non-compliant - customized destructor has empty body
  {
    {
      // Still empty
      ;
    }
  }
};

struct Locker               // Compliant - scoped manager
{
  explicit Locker( std::mutex & m ) :
    m { m }
  {
    m.lock();
  }

  ~Locker()
  {
    m.unlock();
  }

  Locker & operator=( Locker && ) = delete;       // This makes the class unmovable

private:
  std::mutex & m;
};

struct NonMovable           // Non-compliant - copy-constructible, but not
{                           //                 move-constructible
  NonMovable( NonMovable const &  );
  NonMovable( NonMovable       && ) = delete;
};

struct Aggregate { };

struct Child : Aggregate    // Compliant by exception - base class without
{                           //                          destructor is an aggregate
};
```

Rule 15.0.2 *User-provided* copy and move member functions of a class should have appropriate signatures

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

For a `class X`, the copy constructor, move constructor, copy assignment operator and move assignment operator, if *user-provided*, should have the following signatures:

```
X( X const & );                        // Copy constructor
X( X && ) noexcept;                    // Move constructor
X & operator=( X const & ) &;          // Copy assignment operator
X & operator=( X && ) & noexcept;      // Move assignment operator
```


Licensed to: gunjanjain
22 Apr 2025

It is permitted to:

- Add `constexpr` to any of these operations;
- Add `explicit` to the constructors;
- Add `noexcept` (which may be conditional) to the copy operations.

Note: `const X &` is also permitted as an alternative to `X const &`.

Rationale

A constructor taking the class itself by non-const reference parameter (`X &`) is considered to be a copy constructor. However, this parameter style means it is possible to modify the argument object, which would be unlikely to meet developer expectations.

The use of copy and move constructors with parameters having default arguments makes it harder to review the code. Therefore, delegating to constructors supporting these extra parameters should be used in preference to the use of default arguments.

The situation is similar for a copy assignment operator taking the right-hand-side operand by non-const reference. For copy-assignment, the C++ Standard permits the right-hand-side parameter to be pass-by-value; this is not allowed by this rule.

Assignment operators should return an lvalue-reference to the assigned-to object in order to allow chaining of assignments. However, without reference qualification, the assignment may be to a temporary object with the risk that a potentially dangling lvalue-reference to that temporary object may be exposed. Using an lvalue-reference returned from assignment to a temporary object to access the temporary object results in *undefined behaviour* as the temporary object will have been destroyed before the access takes place.

Throwing from within a move operation makes it unclear what the state of the moved-from object is expected to be. Declaring these functions as `noexcept` makes it clear they will not throw, which is compatible with exception-safe code.

Exception

User-provided assignment operators are allowed to be declared with the return type `void` as this prevents use of the result of the assignment operator, easing compliance with Rule 8.18.2.

Example

```
struct UniqueManager
{
  UniqueManager() = default;
  UniqueManager( UniqueManager && ) noexcept;                // Compliant
  UniqueManager & operator=( UniqueManager && ) noexcept;    // Non-compliant -
};                                                           //   needs & qualifier

struct Manager
{
  Manager( Manager const & other ) noexcept( false );     // Compliant
  Manager( Manager const & other, char c );               // Not a copy-constructor
  Manager( Manager && other, char c = 'x' ) noexcept;     // Non-compliant -
};                                                        //   move constructor

struct ScopedManager
{
  ScopedManager();
  ~ScopedManager();
  ScopedManager & operator=( ScopedManager && ) = delete;   // Rule does not apply
};
```


Licensed to: gunjanjain
22 Apr 2025

```
struct Bad
{
  Bad( Bad volatile const & );                  // Non-compliant - volatile
  virtual Bad & operator=( Bad const & ) &;     // Non-compliant - virtual
};
```

See also

Rule 8.18.2, Rule 15.0.1

4.15.1 Constructors [class.ctor]

Rule 15.1.1 An object's dynamic type shall not be used from within its constructor or destructor

[class.abstract] Undefined 6
[class.cdtor] Undefined 3, 4, 5

Category Required

Analysis Undecidable, System

Amplification

For the purposes of this rule, the initialization of a non-static data member (including via a *default member initializer*) is considered as being part of the constructor.

The dynamic type of an object is used when:

- A virtual call is made to a virtual function;
- `typeid` is applied to an object with *polymorphic* `class` type;
- Using `dynamic_cast`.

Rationale

During construction and destruction of an object, its type may be different from the type of the fully constructed object. The result of using an object's dynamic type in a constructor or destructor may not be consistent with developer expectations.

This rule also prohibits a virtual call being made to a pure virtual function from within a constructor or destructor. Such calls result in *undefined behaviour*.

Additionally, using the dynamic type of the current object through a pointer or reference to a child class of the current class results in *undefined behaviour*, and is therefore also prohibited by this rule.

Example

```
class B1
{
public:
  B1()
  {
    typeid( *this );                // Compliant - B1 not polymorphic
  }
};
```


Licensed to: gunjanjain
22 Apr 2025

```
class B2
{
public:
  virtual ~B2();
  virtual void foo();
  virtual void goo() = 0;

  void bar()
  {
    foo();
    typeid( *this );
  }

  B2()
  {
    typeid( *this );                // Non-compliant
    typeid( B2 );                   // Compliant - current object type not used
    B2::foo();                      // Compliant - not a virtual call
    foo();                          // Non-compliant
    goo();                          // Non-compliant - undefined behaviour
    dynamic_cast< B2 * >( this );   // Non-compliant
    bar();                          // Non-compliant - indirect call to foo and
                                    // use of typeid on current object
  }
};
```

The following example is non-compliant and has undefined behaviour when a virtual call is made on the object under construction through an indirect pointer.

```
class B4;

class B3
{
public:
  explicit B3( B4 * b );
  virtual ~B3();
  virtual void foo();
};

class B4 : public B3
{
public:
  B4() : B3( this ) { }
};

B3::B3( B4 * b )
{
    foo();                          // Non-compliant - calls B3::foo
    this->foo();                    // Non-compliant - calls B3::foo
    b->foo();                       // Non-compliant - undefined behaviour
}
```

Rule 15.1.2 All constructors of a class should explicitly initialize all of its virtual base classes and immediate base classes

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to all *user-provided* constructors that are not *defaulted*.

Licensed to: gunjanjain
22 Apr 2025

A base class is considered as explicitly initialized by a constructor when:

1. The base class is initialized in the member initializer list of the constructor; or
2. The constructor is a delegating constructor, assuming that the delegated-to constructor conforms to this rule.

Rationale

This rule reduces the chance of confusion over which constructor will be used, and with what parameters.

Exception

A class is *empty* when it has no non-static data members, no virtual member functions, no virtual base classes, and only *empty* base classes. A base class that is *empty* need not be initialized, as there is nothing to initialize.

Example

```
class A
{
public:
  A() {}                          // Rule does not apply - no base classes
};

class B : public A
{
public:
  B() {}                          // Compliant by exception
};

class V
{
public:
   V() {}                         // Rule does not apply - no base classes
   V( int32_t i ): i ( i ) {}     // Rule does not apply - no base classes

private:
   int32_t i = 0;
};

class C1 : public virtual V
{
public:
  C1() : V { 21 } {}              // Compliant
};

class C2 : public virtual V
{
public:
  C2() : V { 42 } {}              // Compliant
};
```

In the following, there would appear to be an ambiguity as `D` only includes one copy of `V`. Which version of `V`'s constructor is executed and with what parameter? In fact, `V`'s default constructor is always executed. This would be the case even if `C1` and `C2` constructed their bases with the same integer parameter.

```
class D: public C1, public C2
{
public:
  D() {}                          // Non-compliant
};
```


Licensed to: gunjanjain
22 Apr 2025

This is clarified by making the initialization explicit:

```
D() : V {}, C1 {}, C2 {} {}        // Compliant - V::i == 0

struct E
{
  int32_t i;
  int32_t j;
};

class F : public E
{
public:
  F( int32_t val ) : E { val }     // Compliant - E is initialized by aggregate
  {}                               // initialization, with E::j initialized to 0

  F() : F ( 0 ) {}                 // Compliant - delegates to the other constructor
};

class G : public A                 // Rule does not apply - no user-provided
{                                  // constructor
};

class H : public A, public V       // Rule does not apply - no user-provided
{                                  // constructor
public:
  using V::V;                      // Subobject 'A' implicitly initialized
};
```

Rule 15.1.3 Conversion operators and constructors that are callable with a single argument shall be `explicit`

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule does not apply to copy or move constructors.

Note: this rule does not prevent the addition of `explicit` to other constructors.

Rationale

The `explicit` keyword prevents a constructor or conversion operator from being used to implicitly convert from one type to another.

Example

```
class C
{
public:
  C( int32_t a );                    // Non-compliant
};

class D
{
public:
  explicit D( int32_t a );           // Compliant
  D( const D & d );                  // Rule does not apply - copy constructor
  operator int32_t() const;          // Non-compliant
  explicit operator bool() const;    // Compliant
};
```


Licensed to: gunjanjain
22 Apr 2025

```
class E
{
public:
  E( int32_t a,     int32_t b = 0 );  // Non-compliant - callable with one argument
  E( char a = 'a', int32_t b = 0 );  // Non-compliant - callable with one argument
  E( char a,        char b        );  // Rule does not apply - requires two arguments
};

void f( E e );

void g()
{
  f( 0 );                            // Implicit conversion from 0 to E
}
```

Rule 15.1.4 All direct, non-static data members of a class should be initialized before the class object is *accessible*

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

A class object is considered *accessible*:

- At the top of the *compound-statement* that forms the constructor body;
- For an *aggregate*, as soon as the object is created.

A data member is initialized at the top of a constructor body if:

- The constructor is a delegating constructor; or
- The data member has a default member initializer; or
- The data member appears in the constructor's member initialization list; or
- The data member's type has a constructor.

A data member of an *aggregate* is initialized if:

- The data member has a default member initializer; or
- The object's *declaration* has an initializer; or
- The data member's type has a constructor.

For the purposes of this rule, an implicitly or explicitly defaulted constructor is treated as if its synthesized body was user-written.

Rationale

A constructor should completely initialize its object. Explicit initialization reduces the risk of an invalid state existing after successful construction. *Note* — the initialization of base classes is covered by Rule 15.1.2.

Each non-static data member should be initialized, preferably using a default member initializer, or else within a constructor member initialization list.

Assigning to the variable in the constructor body is not sufficient, as requiring members to be initialized at the top of the constructor allows compliance checking for this rule to be made decidable.

Note: compliance with this rule means that constructors will often have an empty body.

Licensed to: gunjanjain
22 Apr 2025

For an *aggregate*, non-static data members can be initialized either by using default member initialization or *aggregate initialization* when declaring an object.

Example

```
class PersonClass
{
public:
  PersonClass( string const & name, int32_t age ) :
    name { name }, age { age }       // Compliant
  {}

  explicit PersonClass( int32_t age ) :
    age { age }                      // Compliant - name is default constructed,
  {}                                 //             and income initialized to 1000

  explicit PersonClass( string const & name ) :
    name { name }                    // Non-compliant - age not initialized
  {
    age = 18;
  }

  PersonClass() = default;           // Non-compliant - age not initialized

private:
  string  name;
  int32_t age;
  int32_t income = 1000;
};

class PersonAggregate
{
public:
  string name;
  int32_t age;
  int32_t income { 1000 };
};

void f()
{
  PersonAggregate p1;       // Non-compliant - age not initialized, even though
                            //                 name and income are initialized
  PersonAggregate p2 {};    // Compliant - name is default constructed, and age is
}                           //             initialized to 0, income to 1000

class Building              // Non-compliant - height not initialized in the
{                           //                 implicit default constructor
private:
  string name;

public:
  int32_t height;
}

class Base
{
  int32_t a;

public:
  explicit Base( int32_t a ) :  // Compliant
    a { a } { }
};
```


Licensed to: gunjanjain
22 Apr 2025

```
class Derived : public Base
{
  int32_t b;

public:
  Derived() :                    // Compliant
    Base { 0 }, b {} {}

  using Base::Base;              // Non-compliant - b not initialized by the
};                               //                 synthesized constructor
```

Rule 15.1.5 A class shall only define an *initializer-list constructor* when it is the only constructor

[dcl.init.list]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Copy and move constructors are permitted in addition to the *initializer-list constructor*.

A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list< T >` or is a reference to a (possibly *cv-qualified*) `std::initializer_list< T >`, and either there are no other parameters or else all other parameters have default arguments.

A constructor is an *initializer-list constructor* if:

- Its first parameter is of type `std::initializer_list< T >` or is a reference to a (possibly *cv-qualified*) `std::initializer_list< T >`; and
- Either there are no other parameters or else all other parameters have default arguments.

Rationale

Under the special overload resolution rules, a constructor with a sole `std::initializer_list< T >` parameter will always be preferred over a constructor taking individual arguments of convertible types in initializations using curly braces. Consequently, the effect of an initialization may differ depending on the form of initialization (curly braces or parentheses) and may not meet developer expectation.

Although the guidelines within this document do not apply to C++ Standard Library definitions, the design of `std::vector` demonstrates the problem that this rule prevents in user classes:

```
std::vector< int32_t > v1{ 3, 4 };    // Vector has two elements: {3, 4}
std::vector< int32_t > v2( 3, 4 );    // Vector has three elements: {4, 4, 4}
```

Another source of confusion arises when a default constructor is present and an object is initialized with empty curly braces. According to language rules, this will always call the default constructor, but a developer may expect an initialization with an empty initializer list.

Note: a non-constructor function taking a single `std::initializer_list< T >` parameter will require both parentheses and curly braces at the call site, so does not suffer from the concern addressed by this rule.

Licensed to: gunjanjain
22 Apr 2025

Example

```
class A                                 // Non-compliant
{
public:
  A( std::size_t x, std::size_t y );

  A( std::initializer_list< std::size_t > list );
};

class B                                 // Compliant - no initializer-list constructor
{
public:
  B( std::size_t x, std::size_t y );

  // The following is not an initializer-list constructor
  B( std::size_t x, std::initializer_list< std::size_t > list);
};

class C                                 // Compliant
{
public:
  C( std::initializer_list< std::size_t > list );

};

class D                                 // Compliant
{
public:
  D( const D & d );
  D( D && d );
  D( std::initializer_list< std::size_t > list );
};
```

4.15.8 Copying and moving class objects [class.copy]

Dir 15.8.1 *User-provided copy assignment operators* and *move assignment operators* shall handle self-assignment

[swappable.requirements]
[moveassignable]
[copyassignable]

Category Required

Amplification

Types supporting copy assignment shall satisfy the *CopyAssignable* requirement.

Types supporting move assignment shall satisfy the *MoveAssignable* requirement.

Additionally, in the case of self-assignment, *user-provided copy assignment operators* and *move assignment operators* shall:

- Not have *undefined behaviour*; and
- Not leak any resources; and
- Preserve the value of the self-assigned object.

Note: what constitutes the value of an object depends on a class's design, and is usually related to the semantics of the equality operator.

Licensed to: gunjanjain
22 Apr 2025

Rationale

Class designs that require *user-provided copy assignment operators* or *move assignment operators* can be avoided when it is possible to use a *manager class* (see Rule 15.0.1), such as smart pointers and the containers provided by the C++ Standard Library. However, when implementing a *manager class*, care needs to be taken when defining *user-provided copy assignment operators* and *move assignment operators*.

Naïve implementations, particularly in the presence of self-assignment, can lead to *undefined behaviour*, resource leaks, performance issues and unintended violations of the object's semantics. Self-assignment is rarely intentional, but it is often hard to spot when it occurs — for example, when manipulating overlapping ranges of objects.

This directive extends the *CopyAssignable* and *MoveAssignable* requirements to all types supporting these assignments, and additionally requires that the state of the object is preserved after a self-assignment. This is done to ensure that the behaviour is predictable and that no resources are leaked.

Well-known idioms, such as copy-and-swap, may help when complying with this directive. However, as there is no one-solution-fits-all, this directive does not recommend a specific idiom.

Example

The following is a simplified implementation of a container, similar to `std::vector`. The class implements a *general manager*, and so *user-provided copy assignment operators* and *move assignment operators* are required.

```
class Vector
{
  std::size_t   size_;
  int32_t     * buffer_;

public:
  Vector() : size_( 0 ), buffer_( nullptr ) {}
  Vector( std::size_t size ) : size_( size ), buffer_( new int[ size ] ) {}

  ~Vector()
  {
    delete[] this->buffer_;
  }

  Vector( Vector const & other ) :
    size_( other.size_ ),
    buffer_( other.size_ != 0 ? new int32_t[ other.size_ ] : nullptr )
  {
    ( void )std::copy_n( other.buffer_, size_, this->buffer_ );
  }

  Vector( Vector  && other ) noexcept :
    size_( std::exchange( other.size_, 0 ) ),
    buffer_( std::exchange( other.buffer_, nullptr ) )
  {}

  Vector & operator=( Vector const & other ) &;
  Vector & operator=( Vector && other ) & noexcept;
};
```


Licensed to: gunjanjain
22 Apr 2025

Copy assignment

The following implementation of copy assignment is non-compliant due to the presence of *undefined behaviour* for self-assignment.

```
Vector & Vector::operator=( Vector const & other ) &
{
  this->size_ = other.size_;
  delete[] this->buffer_;                  // Deletes other.buffer_
  this->buffer_ = new int[ other.size_ ];  // Reading from deleted storage,
                                           //   resulting in undefined behaviour
  ( void )std::copy_n( other.buffer_,
                       other.size_,
                       this->buffer_ );
  return *this;
}
```

This *undefined behaviour* can be prevented by introducing a check for self-assignment.

```
Vector & Vector::operator=( Vector const & other ) &
{
  if ( this != std::addressof( other ) )
  {
    this->size_ = other.size_;
    delete[] this->buffer_;
    this->buffer_ = new int[ other.size_ ];

    ( void )std::copy_n( other.buffer_,
                         other.size_,
                         this->buffer_ );
  }

  return *this;
}
```

The check for self-assignment is a valid solution in this case, but it does not guarantee a correct implementation in all cases (e.g., self-referential data structures). It also has several disadvantages which are outside of the scope of this directive, but which may need to be considered in the final design:

- Pessimization of performance for the (presumably) rare case of self-assignment; and
- Code duplication within the destructor (deletion of elements and the buffer) and the copy constructor (deep copy of elements); and
- Failure to provide the strong exception safety guarantee.

These concerns are addressed when using the copy-and-swap idiom.

```
Vector & Vector::operator=( Vector const & other ) &
{
  Vector tmp( other );                      // Copy construction, with deep copying
  std::swap( this->size_,   tmp.size_   );
  std::swap( this->buffer_, tmp.buffer_ );

  return *this;

  // tmp goes out of scope and thus takes care of deleting the previous buffer
}
```

Self-assignment is handled appropriately when using the copy-and-swap idiom. However, the creation of a new buffer invalidates any iterators or references to elements of the original `Vector`, and it requires unnecessary duplication of resources. Whilst not shown in the above, these issues can be avoided by introducing a check for self-assignment around the copy-and-swap algorithm.

Licensed to: gunjanjain
22 Apr 2025

Move assignment

The following implementation of move assignment has no *undefined behaviour*, but the `Vector` will be released when self-assignment takes place. By any reasonable notion of equivalence, the value is not preserved.

```
Vector & Vector::operator=( Vector && other ) & noexcept
{
  delete[] this->buffer_;
  this->size_   = std::exchange( other.size_,   0       );
  this->buffer_ = std::exchange( other.buffer_, nullptr );

  return *this;
}
```

The following example addresses these issues by using the move-and-swap idiom.

```
Vector & Vector::operator=( Vector && other ) & noexcept
{
    Vector tmp( std::move( other ) );

    std::swap( this->size_,   tmp.size_   );
    std::swap( this->buffer_, tmp.buffer_ );

    return *this;
}
```

When self-assignment takes place, the call to `std::move` transfers ownership of the resources to the temporary object `tmp`, and then the calls to `std:swap` returns their ownership back to `*this`. There are no changes to the state of `*this` and duplication of resources does not occur. Whilst not shown, a check for self-assignment may be included to avoid unnecessary operations.

See also

Rule 15.0.1

4.16 Overloading

4.16.5 Overloaded operators [over.oper]

Rule 16.5.1 The logical AND and logical OR operators shall not be overloaded

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Logical AND and logical OR operators are transformed into function calls. Whilst the overloaded operators obey the rules for syntax and evaluation order defined within the C++ Standard, both operands will always be evaluated. As it may be unclear if a particular use of a logical operator results in a call to an overloaded operator, a developer may incorrectly believe that short-circuit evaluation will occur.

Note: the order of evaluation of the operands was unspecified when using overload operators in versions of C++ prior to C++17.

Licensed to: gunjanjain
22 Apr 2025

Example

In the following example, instantiation of the template function `f` with `AutomatedCar` results in the built-in `operator&&` being used, with `AutomatedCar::increaseSpeed` only being called if `AutomatedCar::isOvertaking` returns *true*.

If `f` is instantiated with `Car`, the overload of `operator&&` is used. As this does not have short-circuit behaviour, `Car::increaseSpeed` is always called, irrespective of the value returned by `Car::isOvertaking`.

```
class FuzzyBool {};

class Car
{
public:
  FuzzyBool isOvertaking();
  bool increaseSpeed();
};

class AutomatedCar
{
public:
  bool isOvertaking();
  bool increaseSpeed();
};

bool operator&&( FuzzyBool fb, bool b );    // Non-compliant

template< class Vehicle >
void f( Vehicle & v )
{
  if ( v.isOvertaking() && v.increaseSpeed() )
  {
  }
}
```

Rule 16.5.2 The *address-of* operator shall not be overloaded

[expr.unary.op] Undefined 5

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Taking the address of an object of incomplete type where the complete type contains a user-declared `operator&` results in *undefined behaviour* (until C++11) or *unspecified behaviour* (since C++11).

Overloading the `&` operator can make code harder to understand as `*&a` may not give the same result as `a`.

Note: `std::addressof` will always return the address of an object without there being a risk of *undefined* or *unspecified behaviour*.

Example

```
// A.h
class A
{
public:
  A * operator&();        // Non-compliant
};
```


Licensed to: gunjanjain
22 Apr 2025

```
// f1.cc
class A;

void f1( A & a )
{
  &a;                     // Undefined or unspecified behaviour
}

// f2.cc
#include "A.h"

void f2( A & a )
{
  &a;                     // Uses user-defined operator&
}
```

4.16.6 Built-in operators [over.built]

Rule 16.6.1 *Symmetrical operators* should only be implemented as non-member functions

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

The following member binary operators are *symmetrical operators*, even when their parameters have different types:

```
operator+    operator-    operator*    operator/    operator%
operator==   operator!=   operator<    operator<=   operator>=   operator>
operator^    operator&    operator|
operator&&   operator||
```

Rationale

This rule helps to ensure that both operands of `a op b` are treated identically in terms of conversions.

If `operator+` for `class C` is implemented as a member (e.g. `C operator+( C rhs ) const;`), then the left-hand value can only be of type `C`, whilst the compiler may implicitly convert right-hand operands of other types to `C`.

For example, if `C` has a constructor that takes an `int` value, then if `c` is a value of type `C`, `c + 1` creates a temporary object from `C( 1 )` and adds it to `c`. Depending upon what other, if any, implicit conversions are available, `1 + c` either results in a compilation error or it may call a different function and give an entirely different result to `c + 1`. This inconsistent behaviour is undesirable.

Note: this rule permits a non-member operator to be declared as a hidden friend (i.e. a friend function defined in the class). Hidden friend operators are only considered for overload resolution by argument-dependent lookup when the compiler has a class object as one of the operands, making it less likely that the wrong overload is selected due to the implicit conversion of both operands. The use of hidden friends for such operators is generally considered to be good practice.

Licensed to: gunjanjain
22 Apr 2025

Example

The constructor in the following example violates Rule 15.1.3.

```
class C
{
  int32_t i;

public:
  C( int32_t x = 0 ): i( x ) {}

  C operator+( C rhs ) const;           // Non-compliant
  C operator+( int32_t rhs ) const;     // Non-compliant
  C operator and( C rhs ) const;        // Non-compliant

  friend C operator*( C lhs, C rhs );   // Compliant - non-member friend

  friend C operator-( C lhs, C rhs )    // Compliant - hidden friend
  {
    return C( lhs.i - rhs.i );
  }

  friend std::ostream &
    operator<<( std::ostream & os,
                      C const & c );    // Rule does not apply - not symmetrical
  C & operator/=( C const & rhs );      // Rule does not apply - not symmetrical
};

C operator/( C lhs, C rhs );            // Compliant - non-member
C operator*( C lhs, C rhs );            // Compliant - non-member friend

int main()
{
  C c( 21 );

  std::cout << (  c + 1 ) << '\n';
//std::cout << (  1 + c ) << '\n';  // Would fail to compile
  std::cout << (  c * 4 ) << '\n';
  std::cout << (  4 * c ) << '\n';
  std::cout << ( 84 / c ) << '\n';
}
```

4.17 Templates

4.17.8 Function template specialization [temp.fct.spec]

Rule 17.8.1 Function templates shall not be explicitly specialized

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule also applies to member function templates, but not non-template member functions of class templates.

Licensed to: gunjanjain
22 Apr 2025

Rationale

Explicit function specializations will be considered only after overload resolution has chosen a best match from the set of primary function templates. Furthermore, when the overload set contains both template and non-template versions that are otherwise an equal match for overload resolution, the template version (and therefore its specializations) will not be selected. All of this may be inconsistent with developer expectations.

Note: overloads provide a better solution than the use of explicit function specializations.

Example

```
template< typename T > void f1( T );       // Overload # 1A
template<> void f1< char * >( char * );    // Non-compliant - explicit
                                           //   specialization of overload # 1A
template< typename T > void f1( T * );     // Overload # 1B

template< typename T > void f2( T   );     // Overload # 2A
template< typename T > void f2( T * );     // Overload # 2B
void f2( char * );                         // Overload # 2C - rule does not apply

template< typename T > void f3( T );
template<> void f3< char * >( char * );    // Non-compliant - explicit
                                           //   specialization of f3

void b( char * s )
{
  f1( s );                                 // Calls overload # 1B, with T = char
  f2( s );                                 // Calls overload # 2C
}
```

4.18 Exception handling

Exceptions provide a way of transferring control to a point in the program higher up in the call stack. They are designed to handle exceptional situations such as unexpected errors.

Code must be designed to behave correctly if exceptions are thrown during execution, otherwise an exception may result in issues such as resource leaks, an invalid program state and unexpected program termination. Compliance with the rules in this section helps to prevent the inappropriate use of exceptions and ensures that the code is more robust if exceptions are raised. In addition, Rule 21.6.2, and more generally the use of the *RAII* idiom, ensures that resources are not leaked in the presence of exceptions.

Like all error handling mechanisms, exceptions have execution time and code size costs. For instance, the stack unwinding that is required for exception handling is often implemented using complex state machines, and most C++ compilers generate code that indirectly invokes `malloc` to allocate heap memory for exception objects.

Due to the complexity of the usual implementations and the allocation of heap memory, it is generally not possible to predict safe and precise upper bounds for the worst-case execution time of exception handling. Therefore, it may be preferable to avoid throwing exceptions in hard real-time code that must guarantee execution deadlines.

Most compilers provide options (such as `-fno-exceptions`) that can be used to disable exceptions in order to eliminate the code and size overheads mentioned above.

Disabling exceptions means that it will probably not be possible to comply with some of the rules within this section and others — for example, it might not be possible to provide the exception handler required by Rule 18.3.1. A number of deviations may therefore be needed, with the supporting documentation being used to demonstrate how the related issues are to be prevented. For example, exceptions that are thrown by the C++ Standard Library will probably lead to immediate program

Licensed to: gunjanjain
22 Apr 2025

termination, and it needs to be demonstrated that any use of the C++ Standard Library will not result in an exception being thrown, or that any termination is handled appropriately.

Note: disabling exceptions is a language extension, requiring a deviation against Rule 4.1.1.

4.18.1 Throwing an exception [except.throw]

Rule 18.1.1 An exception object shall not have pointer type

Category Required

Analysis Decidable, Single Translation Unit

Rationale

If an exception object of pointer type is thrown and that pointer refers to a dynamically created object, then it may be unclear which function is responsible for destroying it, and when. This ambiguity does not exist if the object is thrown by value.

Example

```
class A
{
  // Implementation
};

void fn( int16_t i )
{
  static A   a1;
         A * a2 = new A;

  if ( i > 10 )
  {
    throw &a1;              // Non-compliant - pointer type thrown
  }

  else
  {
    throw a2;               // Non-compliant - pointer type thrown
  }
}
```

See also

Rule 18.3.2

Licensed to: gunjanjain
22 Apr 2025

Rule 18.1.2 An *empty throw* shall only occur within the *compound-statement* of a *catch handler*

[expr.throw]
[except.handle] Implementation 9
[except.terminate] Implementation 2

Category Required

Analysis Decidable, Single Translation Unit

Amplification

An *empty throw* is a *throw-expression* with no operand.

For the purposes of this rule, the body of a lambda declared within the *compound-statement* of a *catch handler* is not considered to be part of the *catch handler*.

Rationale

An *empty throw* re-throws the temporary object that represents an exception. Its use is intended to enable the handling of an exception to be split across two or more handlers.

Syntactically, there is nothing to prevent an *empty throw* from being used outside a *catch handler*. However, this would result in *implementation-defined* program termination when there is no exception object to re-throw.

Example

```
void f1( void )
{
  try
  {
    throw std::range_error( "range error" );
  }

  catch ( std::exception const & )
  {
    log( "Caught in f1" );

    throw;                    // Compliant - re-throws an exception object
  }
}

void f2( void )
{
  throw;                      // Non-compliant - not syntactically within a catch
}
```


Licensed to: gunjanjain
22 Apr 2025

4.18.3 Handling an exception [except.handle]

Rule 18.3.1 There should be at least one exception handler to catch all otherwise unhandled exceptions

[except.terminate] Implementation 1.2, 1.10

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

The function `main` should include a *try-catch* with a `catch ( ... )` handler.

The *catch handlers* of this *try-block*, and any code within `main` that is outside of this *try-block*, should not attempt to propagate an exception. To make this rule decidable, any call to a function having a *potentially-throwing exception specification* ([except.spec]/3) is assumed to propagate an exception.

For the purposes of this rule, where a development environment allows a function other than `main` to be nominated as the entry point of the program, that function shall be treated as if it were `main`.

Rationale

If a program throws an exception that is not caught, the program terminates in an *implementation-defined* manner. In particular, it is *implementation-defined* whether the call stack is unwound before termination, meaning that some destructors may not be executed. By enforcing the provision of a "last-ditch catch-all", the developer can ensure that the program terminates in a consistent manner.

Example

```
int main()                          // Compliant
{
  try
  {
    // Program code
  }

  catch ( specific_type & e )       // Optional, explicit handler(s) are permitted
  {
    // Handle expected exceptions
  }

  catch ( ... )                     // Catch-all handler should be provided
  {
    // Handle unexpected exceptions
  }

  return 0;
}

void logError( char const * message );

int main()                          // Compliant
try
{
  // Program code
}
```


Licensed to: gunjanjain
22 Apr 2025

```
catch ( ... )                        // Catch-all handler
{
  try
  {
    logError( "Unexpected" );
  }

  catch (...)
  {
    // Logging also threw
  }
}

int main()                           // Non-compliant - handler may throw
try
{
  // Program code
}

catch ( ... )                        // Catch-all handler
{
  logError( "Unexpected");           // Potentially throwing function may lead to
}                                    // an exception propagating from main
```

See also

Rule 18.4.1, Rule 18.5.1, Rule 18.5.2

Rule 18.3.2 An exception of `class` type shall be caught by `const` reference or reference

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Slicing occurs if the exception object is of a derived class and it is caught by value as the base class, which means that information unique to the derived class's members is lost. Slicing does not occur when the exception is caught by reference.

Exception objects may be shared between threads, such as when an exception is thrown from `std::shared_future`. In this case, catching by const reference reduces the chance of data races.

Example

```
try
{
  mayThrow();
}
catch ( std::runtime_error e )       // Non-compliant - slicing occurs
{
}
catch ( std::exception const & e )   // Compliant - exception object is complete
{
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 18.3.3 Handlers for a *function-try-block* of a constructor or destructor shall not refer to non-static members from their class or its bases

[except.handle] Undefined 10

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Referring to a non-static member of a class or a base class in the handler (i.e. the *catch* part) of a *function-try-block* of a class constructor or destructor results in *undefined behaviour*.

For example, if a memory allocation exception is thrown during creation of the object, the object will not exist when the handler attempts to access its members. Additionally, in the destructor, the object may have been successfully destroyed before the exception is handled and it will not be available to the handler.

Example

```
class C
{
public:
  int32_t x;

  C()
  try : x { mayThrow() }
  {
  }

  catch ( ... )
  {
    if ( 0 == x )   // Non-compliant - x may not exist at this point
    {
    }
  }

  ~C()
  try
  {
    // Action that may raise an exception
  }

  catch ( ... )
  {
    if ( 0 == x )   // Non-compliant - x may not exist at this point
    {
    }
  }
};
```


Licensed to: gunjanjain
22 Apr 2025

4.18.4 Exception specifications [except.spec]

Rule 18.4.1 *Exception-unfriendly* functions shall be `noexcept`

[support.start.term] Implementation-defined 9.1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The following functions are considered as *exception-unfriendly* and are required to be implicitly or explicitly `noexcept`:

1. Any function or constructor directly called (explicitly or implicitly) to initialize a non-`constexpr`, non-local variable with static or thread storage duration;
2. All destructors;
3. All copy-constructors of an exception object;
4. All move constructors;
5. All move assignment operators;
6. All functions named "swap";

Additionally, the arguments passed to `extern "C"` functions `std::set_terminate`, `std::atexit` or `std::at_quick_exit` shall be convertible to function pointers to `noexcept` functions.

This rule does not apply to any member function defined as `= delete`.

Rationale

When an exception is thrown, the call stack is unwound up to the point where the exception is to be handled. The destructors for all automatic objects declared between the point where the exception is thrown and where it is to be handled will be invoked. If one of these destructors exits with an exception, then the program will terminate in an *implementation-defined* manner. Requiring destructors to be `noexcept` and enforcing Rule 18.5.1 ensures that `std::terminate` does not get called, as required by Rule 18.5.2.

Exceptions from destructors are also undesirable for objects that are at non-local scope or that are declared `static`, as they are destroyed in a "close-down" phase after `main` terminates. There is nowhere within the code that a handler can be placed to catch any exception that may be thrown, leading to a call to `std::terminate`. Similarly, non-local objects may be constructed before `main` starts, meaning that any exception thrown during their construction cannot be caught.

Most destructors are `noexcept` by default, meaning that the omission of an explicit *noexcept-specifier* is generally compliant.

Note: this rule does not apply to the constructors of classes used to construct local objects with static storage duration, as these are constructed the first time their owning function is called (i.e. after `main` has started), allowing exceptions thrown by them to be caught.

When an exception is thrown, the exception object is copy-initialized from the operand of the *throw-expression*. If an exception is thrown during this copy, this is the exception that will be propagated, which may surprise developers. Furthermore, if a *catch handler* catches by value (which is prohibited by Rule 18.3.2), another copy-initialization will happen. If this throws, the program will terminate. It is therefore better to ensure that exception objects' copy constructors do not throw.

Licensed to: gunjanjain
22 Apr 2025

Functions named "swap" are conventionally used as customization points for `std::swap`. The C++ Standard Library containers and algorithms will not work correctly if swapping of two elements exits with an exception.

Non-throwing "swap" functions are also important when implementing the strong exception safety guarantee in a copy (or move) assignment operator. Similarly, move constructors and move assignment operators are usually expected to be non-throwing. If they are not declared `noexcept`, strong exception safety is more difficult to achieve. Furthermore, algorithms may choose a different, possibly more expensive, code path if move operations are not `noexcept`.

Functions passed as arguments to `extern "C"` functions are likely to be invoked from C code that is not able to handle exceptions.

The C++ Standard states that if a function registered using `std::atexit` or `std::at_quick_exit` is called and exits with an exception, then `std::terminate` is called. The C++ Standard requires that a terminate handler set via `std::set_terminate` must not return to its caller, including with an exception (see [terminate.handler]).

Example

```
class C1
{
public:
  C1(){}                          // Compliant - never used at non-local scope

  ~C1(){}                         // Compliant - noexcept by default
};

class C2
{
public:
  C2(){}                          // Not noexcept - see declaration of c2 below
  C2( C2 && other ) {}            // Non-compliant - move constructor

  C2 & operator=( C2 && other );  // Non-compliant - move assignment

  ~C2() noexcept( true ) {}       // Compliant

  friend void swap( C2 &, C2 & ); // Non-compliant - function named swap
};

C2 c2;                            // Non-compliant - construction is non-local

class C3
{
public:
  C3(){}                          // Compliant - c3 in foo not in non-local scope

  ~C3() noexcept( false ) {}      // Non-compliant
};

class MyException : public std::exception       // Non-compliant - implicit copy
{                                               // constructor is noexcept( false )
public:
  MyException ( std::string const & sender );   // Rule does not apply
  const char * what() const noexcept override;

  std::string sender;
};
```


Licensed to: gunjanjain
22 Apr 2025

```
void foo()
{
  static C3 c3;                    // Compliant - constructed on first call to foo

  throw MyException( "foo" );
}

void exit_handler1();              // Non-compliant - passed to atexit

void exit_handler2() noexcept;     // Compliant - also passed to atexit

int main()
{
  try
  {
    const int32_t result1 = std::atexit( exit_handler1 );
    const int32_t result2 = std::atexit( exit_handler2 );
    C1 c1;

    foo();                         // Any exception thrown will be caught below
  }
  catch ( ... ) {}
}

extern "C"
{
  void f( void( *func )() );
}

f( []({} );                        // Non-compliant - function passed to extern "C"
```

See also

Rule 18.3.2, Rule 18.5.1, Rule 18.5.2

4.18.5 Special functions [except.special]

Rule 18.5.1 A *noexcept* function should not attempt to propagate an exception to the calling function

[except.terminate] Implementation 2

Category Advisory

Analysis Undecidable, System

Amplification

A `noexcept` function attempts to propagate an exception when it directly or indirectly throws an exception that is not caught within the function. Any exception that would escape the function causes the program to terminate.

This rule also applies to all functions that are implicitly `noexcept`.

If a function's *exception-specifier* is of the form `noexcept( condition )`, then the function is only permitted to throw an exception when the condition is `false`.

A function's compliance with this rule is independent of the context in which it is called.

Licensed to: gunjanjain
22 Apr 2025

Rationale

Marking a function `noexcept` or `noexcept( true )` does not prevent an exception from being raised in the body of the function. However, if the function attempts to propagate an exception to a calling function, the program will be terminated through a call to `std::terminate`. This results in *implementation-defined* behaviour, including whether or not the stack is unwound before the program terminates (which may result in dangling resources).

Example

```
void mayThrow( int32_t x )
{
  if ( x < 0 )
  {
    throw std::exception {};
  }
}

void notThrowing()
{
}

void f1( int32_t x ) noexcept    // Compliant
{
  notThrowing();
}

void f2( int32_t x ) noexcept    // Compliant
{
  if ( x > 0 )                   // This guard ...
  {
    mayThrow( x );               // ... ensures the call to mayThrow cannot throw
  }
}

void f3( int32_t x ) noexcept    // Non-compliant - even if f3 is only called in
{                                // contexts where x > 0
  mayThrow( x );
}

void f4( int32_t x ) noexcept    // Compliant - any exception will not propagate
{
  try
  {
    mayThrow( x );
  }
  catch( ... )
  {
  }
}

void f5( int32_t x ) noexcept    // Non-compliant - exception is rethrown
{
  try
  {
    mayThrow( x );
  }
  catch ( ... )
  {
    throw;
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

Instantiations of the following template are compliant as they will only be `noexcept( true )` when the addition is non-throwing:

```
template< class T >                                  // Compliant
T plus( T a, T b ) noexcept( noexcept( a + b ) )
{
  return a + b;
}

class A
{
  ~A()
  {
    throw std::exception {};   // Non-compliant - destructor is implicitly noexcept
  }
};

void f6( int32_t x ) throw()   // throw() makes function noexcept
{
  throw std::exception {};     // Non-compliant
}
```

Rule 18.5.2 Program-terminating functions should not be used

[support.start.term] Implementation 3, 5, 9.1, 13
[except.terminate] Implementation 2

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

A program should not contain calls to the C++ Standard Library functions `abort`, `exit`, `_Exit`, `quick_exit` or `terminate`. Additionally, the address of any of these functions should not be taken.

Rationale

If a program terminates due to a call to any of the functions listed above, then the stack will not be unwound and object destructors will not be called. This will potentially leave the environment in an undesirable state (e.g. a file permanently locked).

Taking the address of the functions is not recommended to prevent them from being called via a function pointer.

Notes:

1. This rule only covers explicit calls to the termination functions. The majority of ways in which they may be called implicitly are prevented by Rule 18.1.2, Rule 18.3.1 and Rule 18.5.1.
2. This rule aims to prevent program-terminating functions from being called without the system level implications (such as unreleased resources) being duly considered. If the safety architecture requires rapid termination on the detection of an error, then it may be appropriate to disapply this rule.

Licensed to: gunjanjain
22 Apr 2025

Exception

The call to `abort` that occurs due to the macro expansion of `assert` is not considered to be an explicit call, as it is not expected to be reachable.

Note: a project may consider disallowing this exception if the behaviour of `abort` is not a suitable response to a failed assertion, such as when there is no external mechanism to recover the terminated program.

See also

Rule 18.1.2, Rule 18.3.1, Rule 18.5.1

4.19 Preprocessing directives

4.19.0 MISRA [misra]

Rule 19.0.1 A line whose first token is # shall be a valid preprocessing directive

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to all the lines within a *translation unit*, even if they are excluded by preprocessing.

Note: white-space is permitted before and after the `#` token.

Rationale

A preprocessor directive may be used to conditionally exclude source code until a corresponding `#else`, `#elif` or `#endif` directive is encountered. A malformed or invalid preprocessing directive contained within the excluded source code may not be detected by the compiler, possibly leading to the exclusion of more code than was intended.

Requiring all preprocessor directives to be syntactically valid, even when they occur within an excluded block of code, ensures that this cannot happen.

Example

In the following example all the code between the `#ifndef` and `#endif` directives may be excluded if `AAA` is defined. The developer intended that `AAA` be assigned to `x`, but the `#else` directive was entered incorrectly and not diagnosed by the compiler.

```
#define AAA 2

int32_t foo()
{
  int32_t x = 0;

#ifndef AAA
  x = 1;
#else1        // Non-compliant
  x = AAA;
#endif

  return x;
}
```


Licensed to: gunjanjain
22 Apr 2025

This rule does not apply to the following examples as the # is not a preprocessing token:

```
// Not a preprocessing token within a comment \
#not a token

auto s = R"(
#text)";      // Use in a raw string literal is not a preprocessing token
```

Rule 19.0.2 Function-like macros shall not be defined

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Functions have a number of advantages over function-like macros, including:

- Function arguments and return values are type-checked;
- Function arguments are evaluated once, preventing problems with potential multiple side effects;
- Function names follow classical scoping rules;
- Functions can be overloaded and templatized;
- The address of a function can be passed to another function;
- Function calls can be inlined, providing the same performance characteristics as macros;
- `constexpr` functions can be evaluated at compile-time and may be used in all contexts where a compile-time constant is required;
- In many debugging systems, it is easier to step through execution of a function than a macro.

Exception

As it is not possible to implement equivalent behaviour within a function, a function-like macro may be defined if its definition includes any of the following:

1. `__LINE__`, `__FILE__` or `__func__`;
2. The # or ## operators.

Example

```
#define FUNC( X ) \
  ( ( X ) + ( X ) )             // Non-compliant

template< typename T >
constexpr auto func( T x )      // Possible alternative
{
  return x + x;
}
```

The following examples are compliant by exception:

```
#define ID( name ) \
  constexpr auto name = #name;  // Compliant - use of #

#define TAG( name ) \
  class name##Tag {};           // Compliant - use of ##
```


Licensed to: gunjanjain
22 Apr 2025

```
#define LOG( message ) \
  log( __func__, message );     // Compliant - use of __func__
```

See also

Rule 19.3.1

Rule 19.0.3 `#include` directives should only be preceded by preprocessor directives or comments

[using.headers]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule shall be applied to the contents of a file before preprocessing occurs.

For purposes of this rule, the tokens used to open and close a *linkage-specification* are ignored.

Rationale

To aid code readability, all of the `#include` directives in a particular code file should be grouped together near the top of the file.

Additionally, using `#include` to include a *standard header file* within a declaration or definition, or using part of the C++ Standard Library before the inclusion of the related *standard header file* results in *undefined behaviour*.

Example

```
// f.h
xyz = 0;

// f.cpp
int16_t             // No more includes allowed in f.cpp after this code
#include "f.h"      // Non-compliant

// f1.cpp
#define F1_MACRO
#include "a.h"      // Compliant

extern "C"          // Linkage-specification tokens are ignored
{                   // Linkage-specification token is ignored
  #include "b.h"    // Compliant
}                   // Linkage-specification token is ignored

#include "c.h"      // Compliant

extern "C"          // Linkage-specification tokens are ignored
{                   // Linkage-specification token is ignored
  #include "d.h"    // Compliant

  void g();         // No more includes allowed in f1.cpp after this code
}                   // Linkage-specification token is ignored

#include "e.h"      // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

Rule 19.0.4 `#undef` should only be used for macros defined previously in the same file

[cpp.predefined] Undefined 4

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

Since macros are not subject to the usual scoping rules of the language, complex use of `#undef` can lead to confusion with respect to the existence or meaning of a macro when it is used in the code. However, it might be desirable to limit the number of active macros at any point in the code to help prevent inappropriate use; if a macro is only required for a specific purpose, a common idiom is to `#define` it, use it and `#undef` it immediately afterwards.

Permitting `#undef` to be used for macros that are defined in the same file enables the scope of those macros to be restricted whilst preventing complex uses that could lead to confusion.

Note: undefining a macro defined by the C++ Standard Library can result in *undefined behaviour*.

Example

```
// File.cpp

#include "A.h"        // This header defines the macro M
#undef M              // Non-compliant - defined in another file

#define ID( name ) constexpr auto name = #name

ID( IdA );
ID( IdB );

#undef ID             // Compliant - defined in this file
```

4.19.1 Conditional inclusion [cpp.cond]

Rule 19.1.1 The `defined` preprocessor operator shall be used appropriately

[cpp.cond] Undefined 8

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The only two forms for the `defined` preprocessor operator that are permitted by the C++ Standard are:

```
defined ( identifier )
defined   identifier
```

Generation of the token `defined` during expansion of a macro within a `#if` or `#elif` preprocessing directive is not permitted.

Rationale

Violation of this rule results in *undefined behaviour*.

Licensed to: gunjanjain
22 Apr 2025

Example

```
#if defined 1                                   // Non-compliant - 1 is not an
                                                //                 identifier
#define FEATURE(x) defined(x) && ( x != 0 )
#if FEATURE(X)                                  // Non-compliant - defined resulting
                                                //                 from expansion
#if defined(x) && ( x != 0 )                    // Compliant
```

Rule 19.1.2 All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Confusion can arise when code blocks are included or excluded by the use of conditional compilation directives which are spread over multiple files. Requiring that a `#if` directive be terminated within the same file reduces the visual complexity of the code and the chance that errors will be made during maintenance.

Notes:

1. `#if` directives may be used within included files, provided they are terminated within the same file.
2. It is not clear from the C++ Standard whether such constructs are allowed. Some compilers do require that a `#endif` (etc.) must be in the same file as the associated `#if` (etc.), and all compilers tested raise an error if this is not the case. However, this requirement is not explicitly expressed in the C++ Standard, and there is a reading of the grammar that would allow it. This rule aims to prevent this construct, should any compiler actually allow it.

Example

```
// file1.c
#ifdef A                // Compliant
#include "file1.h"
#endif
// End of file1.c

// file1.h
#if 1                   // Compliant
#endif
// End of file1.h

// file2.c
#if 1                   // Non-compliant
#include "file2.h"
// End of file2.c

// file2.h
#endif
// End of file2.h
```


Licensed to: gunjanjain
22 Apr 2025

Rule 19.1.3 All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be defined prior to evaluation

Category Required

Analysis Decidable, Single Translation Unit

Amplification

As well as using a `#define` preprocessor directive, macro names may effectively be defined in other *implementation-defined*, ways. For example some implementations support:

- Using a compiler command-line option, such as `-D`, to allow macros to be defined prior to translation;
- Pre-defined macros provided by the compiler.

Rationale

If an attempt is made to use an identifier in a preprocessor directive, and that identifier has not been defined as a macro name, then the preprocessor will assume that it has a value of zero. This may not meet developer expectations.

Example

The following example assumes that the identifier `M` is not defined as a macro name.

```
#if M == 0         // Non-compliant
#endif

#if defined( N )  // Compliant - N is not evaluated, even if not a macro
#if N == 0         // Compliant - N is known to be defined at this point
#endif
#endif

// Compliant - B is only evaluated in ( B == 0 ) if it is defined
#if defined( B ) && ( B == 0 )
#endif
```

4.19.2 Source file inclusion [cpp.include]

Rule 19.2.1 Precautions shall be taken in order to prevent the contents of a *header file* being included more than once

Category Required

Analysis Decidable, Single Translation Unit

Amplification

In order to facilitate checking, the contents of the *header file* shall be protected from being included more than once using one of the following two forms of include guard:

```
<start-of-file>
#if !defined ( IDENTIFIER )
#define IDENTIFIER
  // Contents of file
#endif
<end-of-file>
```


Licensed to: gunjanjain
22 Apr 2025

```
<start-of-file>
#ifndef IDENTIFIER
#define IDENTIFIER
  // Contents of file
#endif
<end-of-file>
```

Notes:

1. The identifier used to test and record whether a given *header file* has already been included shall be unique across all the *header files* included within the *translation unit*;
2. Comments are permitted anywhere within these forms.

Rationale

When a *translation unit* contains a complex hierarchy of nested *header files*, it is possible for a particular *header file* to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in erroneous or *undefined behaviour*.

Note: implementations may provide other mechanisms to prevent multiple inclusion — for example `#pragma once` (use of which is restricted by Rule 19.6.1). However, their use is not permitted as they are not specified within the C++ Standard.

Example

```
// file.h
#ifndef FILE_H    // Non-compliant - no include guard in this file
#define FILEH     // <-- this does not #define FILE_H
#endif
```

Rule 19.2.2 The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence

[cpp.include] Undefined 4

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies after macro replacement has been performed.

Rationale

Undefined behaviour occurs if a `#include` directive does not use one of the following forms:

```
#include <filename>
#include "filename"
```

Example

```
#include <filename.h>          // Compliant
#include "filename.h"          // Compliant
#include "./include/cpu.h"     // Compliant - filename may include a path
#include another.h             // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

```
#define HEADER "filename.h"
#include HEADER                // Compliant
#define FILENAME file2.h
#include FILENAME              // Non-compliant

#define BASE "base"
#define EXT  ".ext"
#include BASE EXT              // Non-compliant - expands to an invalid form
                               //   #include "base" ".ext"
                               // - string concatenation occurs after preprocessing
```

Rule 19.2.3 The ' or " or \ characters and the /* or // character sequences shall not occur in a *header file* name

[lex.header] Implementation 2

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Use of the following are *conditionally-supported* with *implementation-defined behaviour*:

- The ' or " or \ characters, and the /* or // character sequences are used between < and > delimiters in a *header file* name preprocessing token;
- The ' or \ characters, or the /* or // character sequences are used between the " delimiters in a *header file* name preprocessing token.

Note: even on systems where \ is the path separator, most implementations will accept the / character as an alternative.

Example

```
#include "file.h"       // Compliant
#include "fi'le.h"      // Non-compliant
#include "path\file"    // Non-compliant
#include "path\\file"   // Non-compliant
#include "path/file"    // Compliant
```

4.19.3 Macro replacement [cpp.replace]

Rule 19.3.1 The # and ## preprocessor operators should not be used

[cpp.stringize] Unspecified 2
[cpp.concat] Unspecified 3

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. It is therefore not always possible to predict the result of macro expansion.

The use of the ## operator can result in code that is hard to understand.

Licensed to: gunjanjain
22 Apr 2025

Note: Rule 4.1.3 covers the *undefined behaviour* that arises if either:

- The result of a # operator is not a valid string literal; or
- The result of a ## operator is not a valid preprocessing token.

See also

Rule 4.1.3, Rule 19.3.2, Rule 19.3.3

Rule 19.3.2 A macro parameter immediately following a # operator shall not be immediately followed by a ## operator

[cpp.stringize] Unspecified 2
[cpp.concat] Unspecified 3

Category Required

Analysis Decidable, Single Translation Unit

Rationale

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. The use of # and ## is discouraged by Rule 19.3.1. In particular, the result of a # operator is a string literal and it is unlikely that pasting this to any other preprocessing token will result in a valid token.

Example

```
#define A( x )       #x          // Compliant
#define B( x, y )    x ## y      // Compliant
#define C( x, y )    x ## #y     // Compliant
#define D( x, y )    #x ## y     // Non-compliant
```

See also

Rule 19.3.1, Rule 19.3.3

Rule 19.3.3 The argument to a *mixed-use macro parameter* shall not be subject to further expansion

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A *mixed-use macro parameter* is a macro parameter that is used both:

- As an operand to # or ##; and
- Not as an operand to these operators.

This rule prohibits invoking a macro with a *mixed-use macro parameter* when the corresponding macro argument is itself subject to further macro replacement.

Rationale

A macro parameter that is used as an operand of a # or ## operator is not expanded prior to being used, whilst the same parameter appearing elsewhere in the replacement text is expanded. This

Licensed to: gunjanjain
22 Apr 2025

causes a macro to exhibit different behavior depending on whether or not its arguments are subject to macro replacement.

Example

In the macro SCALE, X is a *mixed-use macro parameter*.

```
#define SCALE( X ) ( ( X ) * X ## _scale )

int32_t speed;
int32_t speed_scale;

int32_t scaled_speed = SCALE( speed );  // Compliant - speed not expanded
                                        // SCALE expands to
                                        // ( ( speed ) * speed_scale )

#define AA BB
int32_t AA_scale = 1;
int32_t BB = 42;
int32_t BB_scale = 2;

int32_t scaled_AA = SCALE( AA );        // Non-compliant - AA is expanded further
                                        // SCALE expands to ( (BB) * AA_scale )
                                        // User might expect ( (BB) * BB_scale )
```

The rule does not apply to expansions of the following macro as the parameter Y is not a *mixed-use macro parameter*.

```
#define CC( Y ) ( var1 ## Y + var2 ## Y )
```

See also

Rule 19.3.1, Rule 19.3.2

Rule 19.3.4 Parentheses shall be used to ensure macro arguments are expanded appropriately

[Koenig] 78–81

Category Required

Analysis Decidable, Single Translation Unit

Amplification

For the purposes of this rule, a *critical operator* is an operator that has a ranking between 2 and 13 (inclusive), as specified in the table to Rule 8.0.1.

A macro argument containing a *top-level* token (see definition below) that expands as a *critical operator* is inappropriately expanded if, within the macro definition, there is an occurrence of the corresponding macro parameter that is not:

- Directly parenthesized (a parameter x is directly parenthesized in (x)); or
- Stringified (used as an operand to the # operator).

Licensed to: gunjanjain
22 Apr 2025

When a macro is expanded, a *level* can be associated with every token in the expansion of a macro argument. For each argument, the level of its first token is zero, and then the level of each of its subsequent tokens relative to the level of the previous token is:

- One more, if the previous token is `(`
- One less, if the previous token is `)`
- The same, for any other previous token.

A token is said to be *top-level* when its *level* is less than or equal to zero.

Rationale

When a macro is invoked with an argument that looks like an expression, it is generally assumed that this expression will behave as if it were an argument to a function call — in particular, that it will be evaluated in isolation.

However, since macro expansion result in textual replacement, a macro parameter is simply replaced by the text corresponding to the argument. This means that the different tokens that form the argument can end up forming parts of different sub-expressions. This typically happens when the argument contains an operator having a low precedence, and the parameter is expanded next to an operator having a higher precedence. This behaviour can generally be avoided by adding parentheses around the macro parameter.

Example

In the following example, the operator `+` is a *top-level* token in the `x` argument to the macro. However, `x` is neither parenthesized nor stringified in the macro definition. The value of the resulting expression is 7, whereas the value 9 might have been expected.

```
#define M1( x, y ) ( x * y )

r = M1( 1 + 2, 3 );                    // Non-compliant - x not parenthesized
                                       // Expands as r = ( 1 + 2 * 3 );
```

Ideally, the above can be re-written in a compliant manner by parenthesizing the macro parameters in the macro definition:

```
#define M2( x, y ) ( ( x ) * ( y ) )

r = M2( 1 + 2, 3 );                    // Compliant - x is directly parenthesized
                                       // Expands as r = ( ( 1 + 2 ) * ( 3 ) );
```

If this is not possible, it is also acceptable to parenthesize the macro argument:

```
r = M1( ( 1 + 2 ), 3 );                // Compliant - operator + is not top-level
                                       // Expands as r = ( ( 1 + 2 ) * 3 );
```

In the following example, the macro `M1` is invoked with `1 + 2` as its `x` parameter, and the top level `+` token is a *critical operator*. Therefore, `x` is inappropriately expanded, as it is neither parenthesized nor stringified in the macro definition.

```
#define M3( z ) z + 2

r = M1( M3( 1 ), 3 );                  // Non-compliant - operator + is top-level
                                       // Expands as r = ( 1 + 2 * 3 );
```


Licensed to: gunjanjain
22 Apr 2025

Given the macro definition:

```
#define MY_ASSERT( cond )                  \
  do                                       \
  {                                        \
    if ( !cond )                           \
    {                                      \
      std::cerr << #cond << " failed!\n"; \
      std::abort();                        \
    }                                      \
  } while( false )
```

and its use:

```
int32_t x = 0;

MY_ASSERT( x < 42 );  // Non-compliant - argument expansions result in:
                      //   if ( !x < 42 ) - neither parenthesized nor stringified
                      //   "!x < 42"      - stringified
```

During expansion of `MY_ASSERT`, the *cond* parameter is replaced by the argument `x < 42`. This argument includes `<` as a *top-level* token that expands as a *critical operator*, which means that all occurrences of *cond* in the macro definition have to be checked for compliance. Within the macro, *cond* is used:

1. As the operand to `#`, which is compliant as it is stringified; and
2. Within `if( !cond )`, which is non-compliant as it is neither parenthesized nor stringified — the macro expansion will contain `if ( !x < 42 )`, which is *true* for any value of *x* (which is equivalent to `if ( ( !x ) < 42 )`).

Similarly, `MY_ASSERT( a or b )` would also be non-compliant as the rule applies irrespective of the way in which an operator is spelled.

The following example is compliant as the `<` and `>` tokens are not operators in the expanded code.

```
#define PROP( Type, Name ) \
 Type Name;                \
 Type get_##Name() { return Name; }

struct Student
{
  PROP( vector< int32_t >, grades );
}
```

See also

Rule 8.0.1, Rule 19.0.2

Rule 19.3.5 Tokens that look like a preprocessing directive shall not occur within a macro argument

[cpp.replace] Undefined 11

Category Required

Analysis Decidable, Single Translation Unit

Rationale

A macro argument containing sequences of tokens that would otherwise act as preprocessing directives results in *undefined behaviour*.

Licensed to: gunjanjain
22 Apr 2025

Example

```
#define M(A) printf ( #A )

int main()
{
   M(
#ifdef SW        // Non-compliant
   "Message 1"
#else            // Non-compliant
   "Message 2"
#endif           // Non-compliant
    );
}
```

The above could print:

```
#ifdef SW "Message 1" #else "Message 2" #endif
```

or it could print:

```
Message 2
```

or it could exhibit some other behaviour.

4.19.6 Pragma directive [cpp.pragma]

Rule 19.6.1 The `#pragma` directive and the `_Pragma` operator should not be used

[cpp.pragma] Implementation 1
[cpp.pragma.op] 1

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

The effects of the `#pragma` directive and the `_Pragma` operator are *implementation-defined*.

Example

```
#pragma once                    // Non-compliant

#define P( x ) _Pragma( #x )    // Non-compliant

_Pragma( "once" )               // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

4.21 Language support library

4.21.2 Common definitions [support.types]

Rule 21.2.1 The library functions `atof`, `atoi`, `atol` and `atoll` from `<cstdlib>` shall not be used

[C11] / 7.22.1; Undefined 1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

These functions shall not be called or have their addresses taken, and no macro having one of these names shall be expanded.

Note: the same functions from `<stdlib.h>` are also covered by this rule.

Rationale

These functions have *undefined behaviour* associated with them when the string cannot be converted. The C++ library provides safer conversion routines — see [string.conversions], [charconv.from.chars].

Note: [charconv.from.chars] was changed from [utility.from.chars] as the result of a defect report against the C++ Standard.

Example

```
int32_t f( const char * numstr )
{
  return atoi( numstr );          // Non-compliant
}
```

Rule 21.2.2 The *string handling functions* from `<cstring>`, `<cstdlib>`, `<cwchar>` and `<cinttypes>` shall not be used

[cstring.syn]
[cerrno.syn]
[C11] / 7.1.4; Undefined 1
[C11] / 7.24.1; Undefined 1
[C11] / 7.29.4; Undefined 1

Category Required

Analysis Decidable, Single Translation Unit

Amplification

The *string handling functions* are:

From `<cstring>`	`strcat`	`strchr`	`strcmp`	`strcoll`	`strcpy`	`strcspn`
	`strerror`	`strlen`	`strncat`	`strncmp`	`strncpy`	`strpbrk`
	`strrchr`	`strspn`	`strstr`	`strtok`	`strxfrm`	
From `<cstdlib>`	`strtol`	`strtoll`	`strtoul`	`strtoull`	`strtod`	`strtof`
	`strtold`					

Licensed to: gunjanjain
22 Apr 2025

From `<cstring>`	`strcat`	`strchr`	`strcmp`	`strcoll`	`strcpy`	`strcspn`
	`strerror`	`strlen`	`strncat`	`strncmp`	`strncpy`	`strpbrk`
	`strrchr`	`strspn`	`strstr`	`strtok`	`strxfrm`	
From `<cwchar>`	`fgetwc`	`fputwc`	`wcstol`	`wcstoll`	`wcstoul`	`wcstoull`
	`wcstod`	`wcstof`	`wcstold`			
From `<cinttypes>`	`strtoumax`	`strtoimax`	`wcstoumax`	`wcstoimax`		

These functions shall not be called or have their addresses taken, and no macro having one of these names shall be expanded.

Note: the same functions from `<string.h>`, `<stdlib.h>`, `<wchar.h>` and `<inttypes.h>` are also covered by this rule.

Rationale

Incorrect use of some *string handling function* may result in a read or write access beyond the bounds of an object passed as a parameter, resulting in *undefined behaviour*. Also, some *string handling functions* only report errors through the use of `errno`, which is a fragile mechanism — a non-zero value may reveal an error in any function that was called between the last point the value `0` was assigned to `errno` and the current point.

In all cases, the features provided by these functions can be obtained through other C++ Standard Library features which are less error-prone.

Example

```
void f1( char * buffer, size_t bufferSize )
{
  char const * str = "Msg";

  if ( ( strlen( str ) + 1u ) < bufferSize )      // Non-compliant
  {
    ( void ) strcpy( buffer, str );               // Non-compliant
  }
}
```

The following example performs the same operations in a compliant way:

```
void f2( char * buffer, size_t bufferSize )
{
  std::string_view str = "Msg";

  if ( str.size() + 1u < bufferSize )
  {
    str.copy( buffer, str.size() );
    buffer[ str.size() ] = 0;
  }
}
```

See also

Rule 8.7.1

Licensed to: gunjanjain
22 Apr 2025

Rule 21.2.3 The library function `system` from `<cstdlib>` shall not be used

[C11] / 7.22.4.8; Undefined 2; Implementation 2, 3

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This function shall not be called or have its address taken, and no macro having this name shall be expanded.

Note: this rule also applies to `system` from `<stdlib.h>`.

Rationale

The `system` function has *undefined* and *implementation-defined behaviour* associated with it.

Errors related to its use are a common cause of security vulnerabilities.

Rule 21.2.4 The macro `offsetof` shall not be used

[support.types.layout] Undefined 1
[C11] / 7.19; Undefined 3

Category Required

Analysis Decidable, Single Translation Unit

Rationale

The `offsetof` macro is used to access the underlying representation of an object, breaking its encapsulation. In addition, its use results in *undefined behaviour* when the specified member is a bit-field, a static data member, or a member function.

Example

```
struct A
{
  int32_t i;
};

void f1()
{
  offsetof( A, i );      // Non-compliant
}
```

4.21.6 Dynamic memory management [support.dynamic]

C++ allows the construction of objects with dynamic storage duration (see [basic.stc.dynamic]). Dynamic memory may be used explicitly within a program (e.g. sharing or transmitting data across threads in futures and promises), but implicit uses may also occur (e.g. exception handling, containers, run-time type information (RTTI), type erasure with std::function). Because of this, only a limited number of programs are able to completely avoid the use of dynamic memory. In addition, custom memory allocation is not fully supported in all case, making it hard for a program to have complete control over the use of dynamic memory.

This section enumerates a number of potential issues with the use of dynamic memory, and offers some advice on how these can be mitigated within a program.

Licensed to: gunjanjain
22 Apr 2025

4.21.6.1 Terminology

Default allocator

The default definition for the global `new` allocation and `delete` deallocation functions, the `malloc` and `free` functions provided by the system, as well as `std::allocator`.

Custom allocator

Any custom implementation of allocation or deallocation functions, in the form of:

- A replacement definition of the global `new` or `delete` functions; or
- An implementation of the `std::pmr::memory_resource` interface; or
- Some other form.

Local allocator

A custom allocator that is accessible through a local reference to an allocator object, but not via global functions. For example, `std::pmr::monotonic_buffer_resource` is typically used as a local allocator, while the object returned by `std::pmr::new_delete_resource` is not a local allocator.

Fragmentation

Effects which lead to inefficient or wasteful use of memory.

- External fragmentation occurs when free regions of memory are scattered in a way that large contiguous regions of memory become harder or impossible to find. This may prevent an allocation request from succeeding, even though the total amount of free memory is larger than the size requested for the allocation. External fragmentation has a negative effect on execution times.
- Internal fragmentation occurs when an allocator reserves a bigger block of memory than requested, increasing the total amount of consumed memory. The extra memory is not available for use by further allocations and thus is effectively wasted. Internal fragmentation has no direct, negative effect on execution times, and often occurs when using a local allocator that has been designed to give deterministic execution times.

4.21.6.2 Potential safety issues with dynamic memory

Memory management defects

Each dynamically allocated object should be deallocated exactly once and must not be used after deallocation. Failure to do so can lead to a number of undesirable behaviours:

1. Memory leaks — Failure to delete dynamically allocated objects can prevent the system from reclaiming the memory for other uses. While this is not necessarily a critical issue in and of itself, it can lead to resource exhaustion, in particular when leaking memory obtained from a default allocator. Since destructors of leaked objects are usually not executed, leaks can also result in a corruption of the logical program state.
2. Double-delete — Attempting to delete the same memory block twice results in *undefined behaviour*.
3. Passing an invalid pointer to a deallocation function — This results in *undefined behaviour*, and includes the case where a pointer that was obtained from one allocator is passed to the deallocation function of another. Mixing the plain and array forms of the default allocator (`new` or `delete` and `new[]` or `delete[]`) is a common example.
4. Use-after-free — Dereferencing a pointer to an object that has already been deleted results in *undefined behaviour*. Note that dereferencing a pointer to an object with automatic storage duration that has gone out of scope also results in *undefined behaviour*.

Licensed to: gunjanjain
22 Apr 2025

Note: similar issues also apply to other kinds of system-provided resources (e.g. file-handles, locks, ...), and mitigation mechanisms are also necessary when they are used.

Resource exhaustion

The total amount of memory that can be provided by the system is limited. Memory exhaustion errors are difficult to recover from within an application, as the recovery path has to operate under conditions where it is not possible to allocate additional memory.

Certain properties of the default allocator can make memory exhaustion errors more difficult to diagnose. For example, default allocator implementations backed by virtual memory may never fail during memory allocation. Exceeding the amount of available physical memory may not be diagnosed by the allocator, in which case the issue will only surface on the first attempt to access an area of memory that cannot be backed by physical memory. The behaviour will depend on the implementation.

External fragmentation may lead to an allocation failure, as a single block of contiguous memory may not be available, even though the total amount of free memory is still much larger than the amount requested. Internal fragmentation may result in a memory requirement that is larger than the sum of the sizes of the individual allocations — this needs to be taken into account when determining the overall memory requirements of the system.

Non-deterministic execution times

It is generally not possible to determine how long the calls to allocation and deallocation functions will take at runtime:

1. Algorithmic complexity — Many general-purpose allocator implementations are very complex, with runtimes depending on numerous factors that are not known. For example, an allocation call may need to traverse a free list to find a memory block of suitable size, or a deallocation call may have to coalesce multiple, adjacent elements in a free list into a single, contiguous element.
2. Shared global state — The default allocator is a global variable, whose state is manipulated by calls from all over the program. Using the default allocator or a shared allocator object makes it difficult to estimate allocation time at a specific point during program execution.
3. Page allocation — For a hosted application, allocators may need to map additional pages to fulfil an allocation request, requiring additional calls into the operating system (e.g. context switches, syscalls) to provision that memory.
4. Cache behaviour — When using dynamic memory, the layout of objects in memory may change from one run to the next. This can impact execution times as the layout may or may not permit the efficient use of memory caches.

4.21.6.3 Recommended mitigations

Define a lifetime management policy

As for other types of resources, the software design should define a policy for managing the lifetime of allocated memory in order to minimize the risk of memory management defects. Ownership of allocated memory should be clearly assigned to specific entities in the software, for example by the use of *RAII* managing types (smart pointers, containers). The policy should aim to keep ownership relationships as simple as possible, for example, by the use of unique ownership in preference to shared ownership. Note that circular references may prevent memory from being reclaimed when reference counting is used to manage lifetimes (as is the case with `std::shared_ptr`).

Restrict the use of dynamic memory to non-critical phases of execution

The impact of dynamic memory related issues on code can be limited by only allowing calls to allocation or deallocation functions during non-critical phases of program execution — for example, the need for a deterministic execution time is often less strict during startup or shutdown of a system. An application could permit allocation and deallocation calls during the startup phase, but not once the

Licensed to: gunjanjain
22 Apr 2025

system is running. Populating the memory immediately after allocation (which is sometimes known as prefaulting) allows memory exhaustion errors to be detected at the point of allocation instead of at the point of first use.

Use custom allocator implementations

Implementations of general-purpose allocators commonly use complex algorithms that are not optimized for providing reliable worst case execution times. When execution time is critical, it is recommended that a custom allocator be used that relies on an algorithm that makes it easier to reason about worst case execution times and memory consumption. Note that a custom allocator having these attributes may not perform as efficiently in the general case, and it may result in a larger memory footprint when compared to the default allocator. It is also usually preferable to choose an allocator that does not cause external fragmentation.

Use local allocators

The default allocator is a shared global resource, making detection of the issue identified above more difficult. A project should consider the use of local allocators to partition the problem space into smaller subproblems, as these will be easier to analyse in isolation. Note that the use of local allocators has an implicit cost that needs to be considered in the design phase, as their use may considerably increase the complexity of the lifetime management policy. In particular, they introduce new classes of potential lifetime defects, as memory allocations must never outlive their associated allocator, and they must only be deallocated by the allocator they were acquired with.

Rule 21.6.1 *Dynamic memory* should not be used

Category Advisory

Analysis Undecidable, Single Translation Unit

Amplification

Dynamic memory refers to any object with dynamic storage duration that is managed using `operator new` (excluding the non-allocating placement versions), `operator delete`, the functions `calloc`, `malloc`, `realloc`, `aligned_alloc` and `free`, or any platform-specific memory allocation or deallocation function.

Uses of *dynamic memory* may occur implicitly (e.g., when throwing exceptions or using C++ Standard Library classes). Therefore, any instantiation of a C++ Standard Library *entity* having a template argument that is a specialization of `std::allocator` is a violation of this rule, as is any call to a C++ Standard Library function that may use dynamic memory.

Rationale

It is acknowledged that applications may need to use dynamic memory, leading to violations of this rule. Any uses of *dynamic memory* need to be justified through supporting documentation that explains how the issues that have been identified in Section 4.21.6 are managed within the project.

Note: a project may reclassify this rule (see MISRA Compliance [1]) if the risks related to the use of dynamic memory are considered to be unacceptable.

Example

```
auto i = std::make_unique< int32_t >( 42 );    // Non-compliant
auto j = std::vector< int32_t > {};            // Non-compliant
```


Licensed to: gunjanjain
22 Apr 2025

Rule 21.6.2 Dynamic memory shall be managed automatically

[expr.delete]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A program shall not take the address of or use:

1. Any non-placement form of `new` or `delete`;
2. Any of the functions `malloc`, `calloc`, `realloc`, `aligned_alloc`, `free`;
3. Any member function named `allocate` or `deallocate` enclosed by namespace `std`;
4. `std::unique_ptr::release`.

Rationale

The use of dynamic memory requires the tracking of any memory resources that are allocated to ensure that they are released appropriately (no memory leaks, no double frees, use of a matching deallocation function). This is likely to be error prone (possibly leading to *undefined behaviour*) if it is not managed automatically using facilities such as `std::make_unique` or `std::vector`.

In addition, C-style allocation is not type safe and does not invoke constructors or destructors.

Note: the use of placement new, which is non-allocating, is restricted by Rule 21.6.3.

Example

```
class A { /* ... */ };

auto p1 = static_cast< A * >( malloc( sizeof( A ) ) );  // Non-compliant
auto p2 = new A;                                        // Non-compliant
auto p3 = std::make_unique< A >();                      // Compliant
auto p4 = p3.release();                                 // Non-compliant

void f1( std::pmr::memory_resource & mr, A * p )
{
  void * iptr = mr.allocate( sizeof( A ) );             // Non-compliant

  delete p;   // Non-compliant - undefined behaviour if p was allocated using new[]
}
```

See also

Rule 21.6.1, Rule 21.6.3

Licensed to: gunjanjain
22 Apr 2025

Rule 21.6.3 *Advanced memory management* shall not be used

[expr.new]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

All overloads of `operator new` and `operator delete` that are not listed below are *advanced memory management* functions:

```
void * operator new  ( std::size_t count );
void * operator new[]( std::size_t count );
void * operator new  ( std::size_t count, const std::nothrow_t & tag );
void * operator new[]( std::size_t count, const std::nothrow_t & tag );

void operator delete  ( void * ptr ) noexcept;
void operator delete[]( void * ptr ) noexcept;
void operator delete  ( void * ptr, std::size_t sz ) noexcept;
void operator delete[]( void * ptr, std::size_t sz ) noexcept;
void operator delete  ( void * ptr, const std::nothrow_t & tag ) noexcept;
void operator delete[]( void * ptr, const std::nothrow_t & tag ) noexcept;
```

Additionally, `std::launder` and the following functions from the `<memory>` *header file* are also *advanced memory management* functions:

`uninitialized_default_construct`	`uninitialized_default_construct_n`	`destroy`
`uninitialized_value_construct`	`uninitialized_value_construct_n`	`destroy_at`
`uninitialized_copy`	`uninitialized_copy_n`	`destroy_n`
`uninitialized_move`	`uninitialized_move_n`	
`uninitialized_fill`	`uninitialized_fill_n`	

Advanced memory management occurs when:

1. An *advanced memory management* function is either called directly or through a *new-expression* or a *delete-expression*; or
2. The address of an *advanced memory management* function is taken; or
3. A destructor is called explicitly; or
4. Any `operator new` or `operator delete` is user-declared.

Rationale

There are a number of complex issues, such as alignment, object lifetimes and the need to use `std::launder`, that must be considered when using *advanced memory management*. Failure to deal with these appropriately results in the introduction of *undefined behaviour* that is hard to identify.

In addition, *undefined behaviour* results if a user does not provide matching versions of `operator new` and `operator delete`.

These features are generally only used (requiring a deviation) for low-level programming. Ideally, they should be encapsulated to reduce the amount of additional code review that will be required.

Example

```
auto f() noexcept
{
  return new( std::nothrow ) int{ 42 }; // Compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

```
struct X { int32_t a; };

int32_t g()
{
  alignas( X ) std::byte mem[ sizeof( X ) ];

  X * px = new( &mem ) X{ 1 };          // Non-compliant - placement new

  px->~X();                             // Non-compliant - explicit destructor call

  new( px ) X { 2 };                    // Non-compliant - placement new

  return px->a ;                        // Undefined behaviour
}

struct A
{
  void * operator new( size_t );        // Non-compliant
};
```

See also

Rule 21.6.4

Rule 21.6.4 If a project defines either a sized or unsized version of a global `operator delete`, then both shall be defined

[expr.delete]
[new.delete.single]

Category Required

Analysis Decidable, System

Rationale

Within a *delete-expression*, the C++ Standard does not always specify if the sized or the unsized version of the deallocation function will be selected. Therefore, both versions should be provided, and have the same effect, to ensure that the behaviour is well-defined.

Example

The following example is compliant as sized and unsized versions of `operator delete` are provided:

```
void operator delete( void * ptr ) noexcept
{
  std::free( ptr );
}

void operator delete( void * ptr, std::size_t size ) noexcept
{
  delete( ptr );
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 21.6.5 A pointer to an incomplete `class` type shall not be deleted

[expr.delete] Undefined 5

Category Required

Analysis Decidable, Single Translation Unit

Rationale

An incomplete `class` type is a forward declared `class` type for which the compiler has not yet seen a complete definition.

Deleting a pointer to an incomplete `class` type results in *undefined behaviour* when the complete `class` type has a *non-trivial* destructor or a deallocation function.

This rule prohibits deletion of a pointer to an incomplete `class` type even when it is a trivially destructible class without a deallocation function. This restriction defends against a *non-trivial* destructor or a deallocation function being added during development.

Example

The following examples violate Rule 21.6.2.

```
class Bad
{
  class Impl;

  Impl * pImpl;

public:
  ~Bad()
  {
    delete pImpl;   // Non-compliant - at the point of deletion, pImpl points
                    // to an object of incomplete class type.
  }
};

// Header file
class Good
{
  class Impl;

  Impl * pImpl;

public:
  ~Good();
};

// Implementation file
class Good::Impl
{
};

// Good::Impl is a complete type now
Good::~Good()
{
  delete pImpl;   // Compliant - at the point of deletion, pImpl points to
                  // a complete class type.
}
```


Licensed to: gunjanjain
22 Apr 2025

4.21.10 Other runtime support [support.runtime]

Rule 21.10.1 The features of `<cstdarg>` shall not be used

[C11] / 6.9.1; Undefined 8
[C11] / 7.16; Indeterminate 3
[C11] / 7.16.1.1; Undefined 2
[C11] / 7.16.1.2; Undefined 2
[C11] / 7.16.1.3; Undefined 2
[C11] / 7.16.1.4; Undefined 2, 3, 4

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule also applies to the features of `<stdarg.h>`.

None of `va_list`, `va_arg`, `va_start`, `va_end` and `va_copy` shall be used.

Rationale

Passing arguments via an *ellipsis* bypasses the type checking performed by the compiler.

There are many instances of *undefined behaviour* associated with the features of `<cstdarg>`, including:

- `va_end` not being used prior to end of a function in which `va_start` was used;
- `va_arg` being used in different functions on the same `va_list`;
- The type of an argument not being compatible with the type specified to `va_arg`.

Note: this rule does not restrict the use of existing library functions that are implemented as variadic function or the declaration of functions that use the *ellipsis*.

Example

```
#include <cstdarg>

void h( va_list ap )              // Non-compliant
{
  double y;

  y = va_arg( ap, double );       // Non-compliant
}

void f( uint16_t n, ... )
{
  uint32_t x;

  va_list  ap;                    // Non-compliant

  va_start( ap, n );              // Non-compliant
  x = va_arg( ap, uint32_t );     // Non-compliant

  h( ap );

  // Undefined behaviour - ap is indeterminate because va_arg used in h
  x = va_arg( ap, uint32_t );     // Non-compliant

  // Undefined behaviour - returns without using va_end
}
```


Licensed to: gunjanjain
22 Apr 2025

```
void g( void )
{
  // Undefined behaviour - uint32_t / double type mismatch when f uses va_arg
  f( 1, 2.0, 3.0 );
}
```

See also

Rule 8.2.11

Rule 21.10.2 The standard *header file* `<csetjmp>` shall not be used

[csetjmp.syn]
[C11] / 7.13; Undefined 3
[C11] / 7.13.2.1; Undefined 2; Indeterminate 3
[C11] / 7.22.4.4; Undefined 3
[Koenig] 74

Category Required

Analysis Decidable, Single Translation Unit

Amplification

In addition, none of the facilities that are specified as being provided by `<csetjmp>` shall be used.

Note: use of `<setjmp.h>` and the facilities it provides are also prohibited by this rule.

Rationale

The use of `setjmp` and `longjmp` allow the normal function return mechanisms to be bypassed. Their use may result in *undefined* and *unspecified behaviour*. For example, it is *undefined behaviour* if `longjmp` results in the omission of non-trivial object destruction.

Safety standards, such as IEC 61508 [11] (Part 3, Table B.9) or ISO 26262 [9] (Part 6, Table 6), encourage the use of the "single-entry single-exit" principle as part of the "modular approach". Unstructured languages, such as assembly, allow jumps between arbitrary points in a program, violating this principle. C++, with its concept of functions and the corresponding calling mechanism, enforces the "single-entry single-exit" principle through its language definition — for example, multiple return statements within a function all return to the call site.

Note: the C++ Standard states that aspects of the behaviour associated with these facilities are defined in the related version of ISO 9899 [6].

Rule 21.10.3 The facilities provided by the standard *header file* `<csignal>` shall not be used

[C11] / 7.14.1.1; Undefined 3, 5, 7; Implementation 3, 6; Indeterminate 5
[Koenig] 74

Category Required

Analysis Decidable, Single Translation Unit

Amplification

None of the facilities that are specified as being provided by `<csignal>` shall be used.

Note: this rule also applies to the facilities provided by `<signal.h>`.

Licensed to: gunjanjain
22 Apr 2025

Rationale

The inappropriate use of signal handling can lead to *undefined* and *implementation-defined behaviour*.

Note: the C++ Standard states that signal handling behaviour is specified in the related version of ISO 9899 [6].

Exception

Calls to `signal` with a value of `SIG_IGN` as the second (`func`) parameter may be used to disable one or more signals. For example:

```
signal( SIGTERM, SIG_IGN );
```

4.22 Diagnostics library

4.22.3 Assertions [assertions]

Rule 22.3.1 The `assert` macro shall not be used with a *constant-expression*

[dcl.dcl]

Category Required

Analysis Decidable, Single Translation Unit

Rationale

There are a number of limitations to consider when using the `assert` macro. For example:

- An `assert` failure is only reported at run-time, requiring that a failure also has to be managed at run-time;
- The `assert` macro can only be used in contexts where an expression is allowed;
- The `assert` macro may be disabled at build-time.

It is better to use `static_assert` when the condition being asserted is a *constant-expression*, as this ensures that any failure will be detected at compile time.

Exception

`assert( false )` or `assert( false && "any string literal" )` may be used to identify paths that are not expected to be executed.

Example

```
static_assert( ( sizeof( int ) == 4 ), "Bad size" );  // Rule does not apply

void f( int32_t i )
{
  assert( i < 1000 );                                 // Compliant - not constant
```


Licensed to: gunjanjain
22 Apr 2025

```
  if ( i >= 0 )
  {
    assert( ( sizeof( int ) == 4 ) && "Bad size" );   // Non-compliant - constant
  }
  else
  {
    assert( false && "i is negative" );               // Compliant by exception
  }
}
```

4.22.4 Error numbers [errno]

Rule 22.4.1 The literal value zero shall be the only value assigned to `errno`

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Note: the C++ Standard Library is permitted to assign a non-zero value to `errno`.

Rationale

Various functions within the C++ Standard Library set `errno` to a non-zero value to indicate that an error has been detected.

This rule allows this error reporting behaviour to be used, but prevents developers from using `errno` as an error reporting mechanism within a project's code. C++ provides better mechanisms for error handling.

Example

```
std::string getKey ( std::optional< std::string > const & key_data )
{
  if ( key_data.has_value() && !key_data->empty() )
  {
    return key_data.value();
  }

  errno = 42;                  // Non-compliant - non-zero value
  errno = EINVAL;              // Non-compliant - does not expand to literal '0'

  return std::string {};
}

#define OK 0

void f()
{
  uint32_t success { 0 };

  errno = success;             // Non-compliant - must use literal '0'
  errno = OK;                  // Compliant - 'OK' expands to literal '0'

  errnoSettingFunction();

  if ( errno != success )
  {
    handleError();
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

4.23 General utilities library

4.23.11 Smart pointers [smartptr]

Rule 23.11.1 The raw pointer constructors of `std::shared_ptr` and `std::unique_ptr` should not be used

[unique.ptr]
[util.smartptr.shared]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the use of the constructors of `std::shared_ptr` and `std::unique_ptr` that take ownership of the raw pointer passed as an argument.

Rationale

The functions `std::make_shared` and `std::make_unique` perform two operations at the same time:

1. Creating an object dynamically (equivalent to `new`); and
2. Creating a smart pointer that will manage the newly created object's lifetime.

Performing both operations in one step ensures that there is no interleaved operation that could throw an exception before the smart pointer has taken ownership of the object. This also prevents two `unique_ptr` or unrelated `shared_ptr` instances from "owning" the same object.

Notes:

1. `std::make_shared` will allocate a single memory area for both the object and the bookkeeping data required for shared pointers (the reference counts). While this is usually beneficial in terms of performance, it has the drawback that the memory for the object will not be reclaimed when the last `shared_ptr` pointing to it is destroyed, but only when all `weak_ptr` references to the object are also destroyed. If this behaviour is unwanted, a custom variant of `std::make_shared` can be provided that omits this optimisation.
2. Since C++17, the evaluation order of function calls has been made stricter and some of the issues with interleaved calls can no longer happen. However, the use of `make_shared` or `make_unique` is still clearer and can result in better performance.

Example

```
struct A { int8_t i; };
class B { };

void f0()
{
  auto p = std::make_shared< A > ();       // Compliant

  int8_t * pi = &( p->i );
  std::shared_ptr< int8_t > q ( p, pi );  // Does not apply - not taking ownership
}
```


Licensed to: gunjanjain
22 Apr 2025

```
auto f1()
{
  auto * p1 = new A ();
  auto   p2 = std::make_unique< A >();     // make_unique may throw

  return std::shared_ptr< A >( p1 );       // Non-compliant - memory leak if
}                                          //                 make_unique throws

auto f2( std::unique_ptr< A > p )
{
  auto q = p.get();
  // ...
  return std::unique_ptr< A >( q );        // Non-compliant - causes double delete
}

void f3( std::shared_ptr< A > a, std::shared_ptr< B > b );

void f4()
{
  f3( std::shared_ptr< A >( new A() ),
      std::shared_ptr< B >( new B() ) );   // Non-compliant - but well defined
}                                          //                 in C++17
```

Prior to C++ 17, a possible sequencing for the operations in the call to `f3`, where `$n` represents an object in the abstract machine, was:

1. `new A() -> $1`
2. `new B() -> $2`
3. `std::shared_ptr< A >( $1 ) -> $3`
4. `std::shared_ptr< B >( $2 ) -> $4`
5. `f3( $3, $4 )`

If an exception is thrown during the construction of `B`, the object of type `A` will leak. This does not happen in the following as there are no interleaving operations:

```
void f5()
{
  f3( std::make_shared< A >(),
      std::make_shared< B >() );           // Compliant
}
```


Licensed to: gunjanjain
22 Apr 2025

4.24 Strings library

4.24.5 Null-terminated sequence utilities [c.strings]

Rule 24.5.1 The character handling functions from `<cctype>` and `<cwctype>` shall not be used

[C11] / 7.4; Undefined 1
[C11] / 7.30.1; Undefined 5

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the character classification functions and the character case mapping functions from `<cctype>` and `<cwctype>`.

Note: this rule also applies to the same functions from `<ctype.h>` and `<wctype.h>`.

Rationale

The functions declared within `<cctype>` support the classification and case mapping of characters. *Undefined behaviour* occurs if these functions are called with arguments that are not representable as an `unsigned char`, or that are not equal to the value of the macro `EOF`. Similar issues exist for the functions provided by `<cwctype>`.

The C++ Standard Library provides equivalent classification and case mapping functions within `<locale>` that are safer to use.

Note: the C++ Standard states that the behaviour of the functions covered by this rule is defined in the related version of ISO 9899 [6].

Example

```
void f( char c )
{
  if ( std::isdigit( c ) ) {}                    // Non-compliant
  if ( std::isdigit( a, std::locale {} ) ) {}    // Compliant version of the above
}
```

Rule 24.5.2 The C++ Standard Library functions `memcpy`, `memmove` and `memcmp` from `<cstring>` shall not be used

[cstring.syn]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

These functions shall not be called or have their addresses taken, and no macro having one of these names shall be expanded.

Note: this rule also applies to the same functions from `<string.h>`.

Licensed to: gunjanjain
22 Apr 2025

Rationale

Use of `memmove` and `memcpy` can result in *undefined behaviour* if the blocks of memory pointed to by their pointer parameters:

- Overlap (`memcpy` only); or
- Are *potentially-overlapping*; or
- Are not *trivially copyable*.

Additionally, `memcmp` may not indicate equality for objects that are logically equal. Specifically:

- Floating point values may not compare equal, as the floating point format allows multiple representations for some values, such as zero and minus zero (which will not compare equal); and
- Class objects may not compare equal due to:
  - Padding between members, as its content is unspecified and effectively indeterminate; or
  - Unions having different active members or members of different sizes.
- Buffers may not compare equal when the meaningful content does not occupy the whole buffer and the whole buffer is compared. For example, this may happen with:
  - `std::vector`, where memory is preallocated to enable efficient growth; or
  - C-style strings, where the `\0` terminator may occur within the buffer and be followed by irrelevant data.

Example

```
void f1( const uint8_t * src, uint8_t * dst, size_t len )
{
  memmove( dst, src, len );                                  // Non-compliant
}

struct S
{
  bool m1;
  // There may be padding here
  int64_t m2;
};

void f2( S s1, S s2 )
{
  if ( memcmp( &s1, &s2, sizeof( S ) ) != 0 )                // Non-compliant
  {
  }
};

extern char buffer1[ 12 ];
extern char buffer2[ 12 ];

void f3()
{
  strcpy( buffer1, "abc" );      // Indeterminate contents in elements 4 to 11
  strcpy( buffer2, "abc" );      // Indeterminate contents in elements 4 to 11

  if ( memcmp( buffer1, buffer2, sizeof( buffer1 ) ) != 0 )   // Non-compliant
  {
  }
}
```


Licensed to: gunjanjain
22 Apr 2025

4.25 Localization library

4.25.5 C library locales [c.locales]

Rule 25.5.1 The `setlocale` and `std::locale::global` functions shall not be called

[locale] Undefined 9

Category Required

Analysis Decidable, Single Translation Unit

Rationale

Calls to `setlocale` or `std::locale::global` may introduce data races (leading to *undefined behaviour*) with functions that use the locale (e.g. `printf`, `tolower`). It is not as easy to guard against these potential data races due to the ways in which the global locale is used within the C++ Standard Library.

The C++ Standard Library provides functions that allow a locale to be passed as an argument, meaning that it is possible to use a specific locale without having to depend on the setting of the global locale objects.

Example

```
void f1()
{
  wchar_t c = L'\u2002';                    // En-space

  std::setlocale( LC_ALL, "ja_JP.utf8" );   // Non-compliant

  if ( std::isspace( c ) ) {}               // Uses global locale
}
```

The following example sets the locale without violating this rule:

```
void f2()
{
  wchar_t c = L'\u2002';                    // En-space

  std::locale utf8( "ja_JP.utf8" );

  if ( std::isspace( c, utf8 ) ) {}         // Does not use global locale
}
```


Licensed to: gunjanjain
22 Apr 2025

Rule 25.5.2 The pointers returned by the C++ Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` must only be used as if they have pointer to const-qualified type

[clocale.syn]
[cstdlib.syn]
[cstring.syn]
[C11] / 7.11.1.1; Undefined 8
[C11] / 7.11.2.1; Undefined 8
[C11] / 7.22.4.6; Undefined 4
[C11] / 7.24.6.2; Undefined 3

Category Mandatory

Analysis Decidable, Single Translation Unit

Amplification

The `localeconv` function returns a pointer of type `struct lconv *`. This pointer must be regarded as if it had type `const struct lconv *`.

A `struct lconv` object includes pointers of type `char *` and the `getenv`, `setlocale`, and `strerror` functions each return a pointer of type `char *`. These pointers are used to access C-style strings (null-terminated arrays of type *char*). For the purposes of this rule, these pointers must be regarded as if they had type `const char *`.

The addresses of these functions shall not be taken.

Rationale

The C++ Standard states that *undefined behaviour* occurs if a program modifies:

- The structure pointed to by the value returned by `localeconv`;
- The strings returned by `getenv`, `setlocale` or `strerror`.

Note: the C++ Standard does not specify the behaviour that results if the strings referenced by the structure pointed to by the value returned by `localeconv` are modified. This rule prohibits any changes to these strings as they are considered to be undesirable.

Treating the pointers returned by the various functions as if they were const-qualified allows an analysis tool to detect any attempt to modify an object through one of the pointers. Additionally, assigning the return values of the functions to const-qualified pointers will result in the compiler issuing a diagnostic if an attempt is made to modify an object.

Note: if a modified version is required, a program should make and modify a copy of any value covered by this rule.

Preventing the addresses of these functions from being taken allows compliance checks to be made decidable.

Licensed to: gunjanjain
22 Apr 2025

Example

The following examples are non-compliant as the returned pointers are assigned to non const-qualified pointers. Whilst this will not be reported by a compiler (it is not *ill-formed*), an analysis tool will be able to report a violation.

```
void f1()
{
  char         * s1   = setlocale( LC_ALL, 0 );   // Non-compliant
  struct lconv * conv = localeconv();             // Non-compliant

  s1[ 1 ]             = 'A';                      // Undefined behaviour
  conv->decimal_point = "^";                      // Undefined behaviour
}
```

The following examples are compliant as the returned pointers are assigned to const-qualified pointers. Any attempt to modify an object through a pointer will be reported by a compiler or analysis tool as this is *ill-formed*.

```
void f2()
{
  char str[ 128 ];

  ( void ) strcpy( str,
                   setlocale( LC_ALL, 0 ) );      // Compliant - 2nd parameter to
                                                  // strcpy takes a const char *
  const struct lconv * conv = localeconv();       // Compliant

  conv->decimal_point = "^";                      // Ill-formed
}
```

The following example shows that whilst the use of a const-qualified pointer gives compile time protection of the value returned by `localeconv`, the same is not true for the strings it references. Modification of these strings can be detected by an analysis tool.

```
void f3()
{
  const struct lconv * conv = localeconv();       // Compliant

  conv->grouping[ 2 ] = 'x';                      // Non-compliant
}
```

Rule 25.5.3 The pointer returned by the C++ Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` must not be used following a subsequent call to the same function

Category Mandatory

Analysis Undecidable, System

Amplification

Calls to `setlocale` may change the values accessible through a pointer that was previously returned by `localeconv`.

Licensed to: gunjanjain
22 Apr 2025

For the purposes of this rule:

- A call to `setlocale` following a call to `localeconv` are treated as if they are calls to the same function; and
- Calls to `asctime` and `ctime` are treated as if they are calls to the same function; and
- Calls to `gmtime` and `localtime` are treated as if they are calls to the same function.

Note: calls to `setlocale` or `localeconv` within a different thread of execution may lead to violations of this rule.

Rationale

The C++ Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` and `strerror` return a pointer to an object within the library's implementation. The implementation is permitted to use static buffers for any of these objects and a second call (which may occur in a different thread) to the same function may modify the contents of the buffer. The value accessed through a pointer held by the program before a subsequent call to a function may therefore change unexpectedly.

Note: the C++ Standard states that the behaviour of the functions covered by this rule is specified in the related version of ISO 9899 [6].

Example

```
void f1()
{
  const struct lconv * lc = localeconv();

  std::string copy { lc->int_curr_symbol };

  const char * res = std::setlocale ( LC_MONETARY, "fr_FR" );

  std::cout << lc->int_curr_symbol;  // Non-compliant - use after setlocale called
  std::cout << copy;                 // Compliant - copy made before call
  std::cout << res;                  // Compliant - no subsequent call before use
}
```

See also

Rule 25.5.1

4.26 Containers library

4.26.3 Sequence containers [sequences]

Rule 26.3.1 `std::vector` should not be specialized with `bool`

[container.requirements.dataraces]

Category Advisory

Analysis Decidable, Single Translation Unit

Rationale

The `std::vector< bool >` specialization's behaviour differs from that of other uses of `std::vector` as it uses optimized space allocation. For example, the `data` member function is not available.

Licensed to: gunjanjain
22 Apr 2025

The C++ Standard guarantees that, in general, elements of a C++ Standard Library container can be modified concurrently, but specifically notes that this is not true for `std::vector< bool >`.

Note: other C++ Standard Library containers do not have specializations for `bool` and do not exhibit the behaviours identified above.

Example

```
struct myBool { bool b; };          // Wrapper for bool

void foo() noexcept
{
  std::vector< bool         > a;    // Non-compliant - optimized storage
  std::vector< std::uint8_t > b;    // Compliant
  std::vector< myBool       > c;    // Compliant
  std::array < bool, 20     > d;    // Rule does not apply
  std::bitset< 200          > e;    // Rule does not apply - efficient storage
}
```

4.28 Algorithms library

4.28.3 Algorithms requirements [algorithms.requirements]

Rule 28.3.1 *Predicates* shall not have *persistent side effects*

[algorithms.requirements]
[alg.sorting]

Category Required

Analysis Undecidable, System

Amplification

When a template parameter is named `Compare`, `Predicate`, or `BinaryPredicate` in the C++ Standard Library, every callable passed as an argument of that type is a *predicate*.

In addition to not having *persistent side effects*, if the *predicate* is a *function object*, its `operator()` shall be declared `const`.

Note: the `operator()` of a lambda closure is `const` unless the lambda is declared `mutable`.

Rationale

It is *implementation defined* if the *predicate* used by an algorithm will be copied. The state of such a *predicate* may therefore unexpectedly be different if a copy is made. Additionally, most algorithms do not specify in which order the *predicates* will be invoked, or on which objects. This makes it very difficult to implement a sensible *predicate* with mutable internal state.

Ideally, a tool will provide a mechanism that allows the identification of additional *predicates* in order to include them in the analysis scope of this rule.

Licensed to: gunjanjain
22 Apr 2025

Example

```
bool bar( std::vector< int32_t > & v, int32_t & count )
{
  return std::any_of( v.begin(), v.end(),
    [&count]( int32_t i )                                  // Non-compliant
    {
      if ( i == 3 )
      {
        ++count;                                           // Persistent side effect
        return true;
      }
      return false;
    } );
}

struct Comp
{
  bool operator()( int32_t a, int32_t b )                  // Non-compliant - not const
  {
    return a > b;
  }
};

std::set< int32_t, Comp > mySet;
```

4.28.6 Mutating sequence operations [alg.modifying.operations]

Rule 28.6.1 The argument to `std::move` shall be a non-const *lvalue*

[forward]
[class.copy.ctor]
[class.copy.assign]

Category Required

Analysis Decidable, Single Translation Unit

Rationale

The result of calling `std::move` on an object that is `const` will result in the object's content not being moved.

Calling `std::move` on an *rvalue* is redundant.

Example

```
void f1( std::string && );                    // #1
void f1( std::string const & );               // #2

void f2( std::string const & s1, std::string s2 )
{
  f1( s1 );                                   // Calls #2
  f1( std::move( s1 ) );                      // Non-compliant - calls #2

  f1( std::move( s2 ) );                      // Compliant - calls #1

  f1( std::string( "abc" ) );                 // Calls #1
  f1( std::move( std::string( "abc" ) ) );    // Non-compliant - redundant move of
}                                             // temporary, also calls #1
```


Licensed to: gunjanjain
22 Apr 2025

Rule 28.6.2 *Forwarding references* and `std::forward` shall be used together

[dcl.ref]
[temp.deduct.call]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

A *forwarding reference* parameter (of type `T &&`) shall be forwarded when passed to other functions by wrapping the parameter in a call to the function `std::forward< T >`.

Furthermore, `std::forward` shall only be used to forward a *forwarding reference*.

Rationale

Perfect forwarding relies on language features such as reference collapsing and type deduction, which are complex to master. Enforcing the use of well known idioms avoids the risk of writing code that does not do what was intended.

Note: care must be taken not to forward the same argument twice — see Rule 28.6.3.

Example

```
void f1( std::string &  );          // #1
void f1( std::string && );          // #2

template< typename T1, typename T2 >
void f2( T1 && t1, T2 & t2 )
{
  f1( t1 );                         // Non-compliant - calls #1

  f1( std::forward< T1 >( t1 ) );   // Compliant - calls #1 (for #4) or #2 (for #3)
  f1( std::forward< T2 >( t2 ) );   // Non-compliant - calls #2
  f1( std::forward< T2 >( t1 ) );   // Non-compliant - wrong template parameter

  f1( std::move( t1 ) );            // Non-compliant - calls #2
  f1( std::move( t2 ) );            // Rule does not apply - calls #2

  auto lambda = [] ( auto && t )
                  { f1(t); };       // Non-compliant - calls #1
}

void f3()
{
  std::string s;

  f2( std::string { "Hello" }, s ); // #3
  f2( s, s );                       // #4
}
```


Licensed to: gunjanjain
22 Apr 2025

```
template< typename T >
struct A
{
  void foo( T && t )
  {
    std::move( t );    // Rule does not apply - not a forwarding reference
  }
};
```

See also

Rule 28.6.1, Rule 28.6.3

Rule 28.6.3 An object shall not be used while in a *potentially moved-from state*

Category Required

Analysis Decidable, Single Translation Unit

Amplification

Calling `std::move`, `std::forward` or using an equivalent `static_cast` puts its argument into a *potentially moved-from state*.

An object in a *potentially moved-from state* shall not be used on any path, regardless of the path's feasibility.

An object passed as an *lvalue reference* function parameter shall not be in a *potentially moved-from state* when the function returns. This additional restriction is included as it allows compliance to be determined within a *translation unit*.

This rule does not apply to the following:

- Assigning to an object; or
- Destroying an object; or
- Using an object having type `std::unique_ptr`.

For the purposes of this rule, aliases of an object are considered to refer to different objects. This allows compliance checks to be decidable.

Rationale

Using `std::forward` or `std::move` on an *lvalue* to pass it as an *rvalue reference* argument in a function call can result in the *lvalue* object being in an indeterminate state after the call. However, a `std::unique_ptr` that has been *moved-from* is in a well-defined state, equal to `nullptr`.

Example

```
size_t a( std::string s1 )
{
  std::string s2 = std::move( s1 );
  return s1.size();                      // Non-compliant - s1 has potentially
}                                        //                 moved-from state

size_t b( std::string s1 )
{
  std::string s2 =
    static_cast< std::string && >( s1 ); // Equivalent to std::move
  return s1.size();                      // Non-compliant - s1 has potentially
}                                        //                 moved-from state
```


Licensed to: gunjanjain
22 Apr 2025

```
void c( std::string s1 )
{
  std::string s2 = std::move( s1 );
  std::string s3 = s1;                    // Non-compliant - s1 has potentially
}                                         //                 moved-from state

template< typename T >
void bar( T & t );

template< typename T >
void foo( T && t )
{
  bar( std::forward< T >( t ) );
  ++t;                                    // Non-compliant - std::forward leaves t
}                                         //   in a potentially moved-from state

struct X { std::string s; };

void f( X & x )
{
  X y ( std::move( x ) );                 // Non-compliant - lvalue reference
                                          //   parameter left in potentially moved-
}                                         //   from state when function returns

void g( X x )
{
  X y;

  y = std::move( x );                     // Compliant - no more uses of x
}

void h( X x )
{
  X y;

  y = std::move( x );
  x = X{};                                // Compliant - assigns to potentially
}                                         //             moved-from object
```

The following is non-compliant as the evaluation order of the arguments to `d1` is *implementation-defined* and there is a permitted order in which the first argument `s` has *potentially moved-from state*.

```
void    d1 ( std::string const &, int32_t );
int32_t d2 ( std::string && );

void d3( std::string s )
{
  d1( s, d2( std::move( s ) ) );          // Non-compliant
}
```

Rule 28.6.4 The result of `std::remove`, `std::remove_if`, `std::unique` and `empty` shall be *used*

[container.requirements.general]
[alg.modifying.operations]

Category Required

Analysis Decidable, Single Translation Unit

Amplification

For the purposes of this rule, a cast to `void` is not considered to be a *use*.

Licensed to: gunjanjain
22 Apr 2025

This rule applies to member and non-member forms of `empty` within the C++ Standard Library.

This rule does not apply to the `std::remove( const char * )` overload defined in `<cstdio>`.

Rationale

The mutating algorithms `std::remove`, `std::remove_if` and overloads of `std::unique` operate by swapping or moving elements of the range they are operating over. On completion, they return an iterator to one past the last valid element. In the majority of cases, the correct behaviour is to use this result as the first operand in a call to `std::erase`.

Ignoring the result of `empty` may indicate that a developer expects the call to purge the contents of the container, while it actually reports if it contains data.

Example

```
void f1()
{
  std::vector< int32_t > v1 = { 0, 0, 1, 1, 2, 2, 3, 3 };
  std::vector< int32_t > v2 = v1;

  std::unique( v1.begin(), v1.end() );           // Non-compliant

  // v1 now holds { 0, 1, 2, 3, ?, ?, ?, ? }
  // where ? represents an unknown value

  v2.erase( std::unique( v2.begin(), v2.end() ),
                         v2.end() );             // Compliant

  // Contents of v2 is now { 0, 1, 2, 3 }
}

void f2( std::vector< int32_t > & v3 )
{
  empty( v3 );                                   // Non-compliant - result not used
  v3.empty();                                    // Non-compliant - result not used

  if ( !empty( v3 ) )                            // Compliant
  {
    v3.clear();                                  // Purges the vector
  }
}
```

See also

Rule 0.1.2

4.30 Input/output library

4.30.0 MISRA [misra]

Rule 30.0.1 The C Library input/output functions shall not be used

Category Required

Analysis Decidable, Single Translation Unit

Amplification

This rule applies to the functions that are specified as being provided by `<cstdio>` and the wide-character equivalents specified as being provided by `<cwchar>`.

Licensed to: gunjanjain
22 Apr 2025

None of these identifiers shall be used and no macro with one of these names shall be expanded.

Notes:

1. Use of the same functions from `<stdio.h>` and `<wchar.h>` are also prohibited by this rule.
2. This rule does not prohibit the use of the facilities provided by `<fstream>`, even though they may indirectly use functions from `<cstdio>` or `<cwchar>`.

Rationale

Streams and file input/output have *undefined*, *unspecified* and *implementation-defined behaviours* associated with them.

Rule 30.0.2 Reads and writes on the same file stream shall be separated by a positioning operation

[filebuf]
[C11] / 7.21.5.3; Undefined 7

Category Required

Analysis Undecidable, System

Amplification

An explicit, interleaving stream positioning operation shall be used between input operations and output operations on a `std::basic_filebuf`.

This rule applies to direct and indirect calls (e.g. from `std::fstream`) to `std::basic_filebuf`.

Note: for the purposes of this rule, a call to `fflush` after an output operation is considered to be an explicit file positioning operation.

Rationale

The C `FILE *` abstraction, used as the underlying system file input/output for `std::basic_filebuf`, holds a single file position that is used when reading from or writing to the file. Using an input operation on a `FILE *` immediately after an output operation (or vice versa) results in *undefined behaviour*, unless an interleaving file positioning operation is used to update the file's position.

In addition, a `streambuf` object keeps separate buffer positions for reading and writing characters from its internal buffer. A `basic_filebuf` object is only guaranteed to synchronize the separate internal `streambuf` read and write positions that it maintains when a positioning operation is called when alternating between reading and writing (and vice versa). Failure to include such a positioning operation leads to *undefined behaviour*.

The accessible positioning operations for `streambuf` are `pubseekoff` and `pubseekpos`, whilst for file streams they are `tellg`, `seekg`, `tellp`, and `seekp`. One of these functions shall be called when switching from output to input, or vice versa.

Licensed to: gunjanjain
22 Apr 2025

Example

```
void show_fstream_non_compliant()
{
  std::fstream f { "hello.txt" };

  f << "Hello world!\n" << std::flush;   // flush is not a positioning operation

  std::string s {};

  std::getline( f, s );                  // Non-compliant - undefined behaviour
}

void show_fstream_compliant()
{
  std::fstream f { "hello.txt" };

  f << "Hello world!\n";

  std::string s {};

  f.seekg( 0, std::ios_base::beg );

  std::getline(f, s);                    // Compliant - s holds "Hello world!"
}
```


Licensed to: gunjanjain
22 Apr 2025

5 References

5.0 MISRA publications

[1] MISRA Compliance:2020, *Achieving compliance with MISRA coding guidelines*
ISBN 978-1-906400-26-2
HORIBA MIRA Limited, February 2020

[2] MISRA C++:2008, *Guidelines for the use of the C++ language in critical systems*
ISBN 978-1-906400-03-3
MIRA Limited, June 2008

[3] MISRA *Development guidelines for vehicle based software*
ISBN 0-9524156-0-7
Motor Industry Research Association, November 1994

[4] MISRA AC GMG:2023, *Generic modelling design and style guidelines*
ISBN 978-1-911700-04-3 Paperback, ISBN 978-1-911700-05-0 PDF
The MISRA Consortium Limited, June 2023

[5] MISRA AC SLSF:2023, *Modelling design and style guidelines for the application of Simulink and Stateflow*
ISBN 978-1-911700-06-7 Paperback, ISBN 978-1-911700-07-4 PDF
The MISRA Consortium Limited, June 2023

5.1 International standards

[6] ISO/IEC 9899:2011, *Programming languages — C*
International Organization for Standardization

[7] ISO/IEC 10646:2020, *Information technology — Universal coded character set (UCS)*
International Organization for Standardization

[8] ISO/IEC 14882:2017, *Information technology — Programming languages — C++*
International Organization for Standardization

[9] ISO 26262:2018, *Road vehicles — Functional safety*
International Organization for Standardization

[10] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-Point arithmetic*
International Organization for Standardization

[11] IEC 61508:2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems*
International Electromechanical Commission

[12] DO-178C/ED-12C, *Software Considerations in Airborne Systems and Equipment Certification*
RTCA/EUROCAE, 2011

[13] Unicode 13.0.0, *Unicode Standard Annex #44*
Unicode, Inc.

Licensed to: gunjanjain
22 Apr 2025

5.2 Other references

[14] Koenig A., *C Traps and Pitfalls*
ISBN 0-201-17928-8
Addison-Wesley, 1988

Licensed to: gunjanjain
22 Apr 2025

Appendix A Summary of guidelines

Language independent issues

Path feasibility [misra]

Rule 0.0.1	Required	A function shall not contain unreachable statements
Rule 0.0.2	Advisory	Controlling expressions should not be invariant

Unused values [misra]

Rule 0.1.1	Advisory	A value should not be unnecessarily written to a local object
Rule 0.1.2	Required	The value returned by a function shall be used

Unused declarations [misra]

Rule 0.2.1	Advisory	Variables with limited visibility should be used at least once
Rule 0.2.2	Required	A named function parameter shall be used at least once
Rule 0.2.3	Advisory	Types with limited visibility should be used at least once
Rule 0.2.4	Advisory	Functions with limited visibility should be used at least once

Runtime failures [misra]

Dir 0.3.1	Advisory	Floating-point arithmetic should be used appropriately
Dir 0.3.2	Required	A function call shall not violate the function's preconditions

General principles

Implementation compliance [intro.compliance]

Rule 4.1.1	Required	A program shall conform to ISO/IEC 14882:2017 (C++17)
Rule 4.1.2	Advisory	Deprecated features should not be used
Rule 4.1.3	Required	There shall be no occurrence of undefined or critical unspecified behaviour

Program execution [intro.execution]

Rule 4.6.1	Required	Operations on a memory location shall be sequenced appropriately

Lexical conventions

MISRA [misra]

Rule 5.0.1	Advisory	Trigraph-like sequences should not be used

Comments [lex.comment]

Rule 5.7.1	Required	The character sequence /* shall not be used within a C-style comment

Licensed to: gunjanjain
22 Apr 2025

Dir 5.7.2	Advisory	Sections of code should not be “commented out”
Rule 5.7.3	Required	Line-splicing shall not be used in // comments

Identifiers [lex.name]

Rule 5.10.1	Required	User-defined identifiers shall have an appropriate form

Literals [lex.literal]

Rule 5.13.1	Required	Within character literals and non raw-string literals, \ shall only be used to form a defined escape sequence or universal character name
Rule 5.13.2	Required	Octal escape sequences, hexadecimal escape sequences and universal character names shall be terminated
Rule 5.13.3	Required	Octal constants shall not be used
Rule 5.13.4	Required	Unsigned integer literals shall be appropriately suffixed
Rule 5.13.5	Required	The lowercase form of L shall not be used as the first character in a literal suffix
Rule 5.13.6	Required	An integer-literal of type long long shall not use a single L or l in any suffix
Rule 5.13.7	Required	String literals with different encoding prefixes shall not be concatenated

Basic concepts

MISRA [misra]

Rule 6.0.1	Required	Block scope declarations shall not be visually ambiguous
Rule 6.0.2	Advisory	When an array with external linkage is declared, its size should be explicitly specified
Rule 6.0.3	Advisory	The only declarations in the global namespace should be main, namespace declarations and extern "C" declarations
Rule 6.0.4	Required	The identifier main shall not be used for a function other than the global function main

One-definition rule [basic.def.odr]

Rule 6.2.1	Required	The one-definition rule shall not be violated
Rule 6.2.2	Required	All declarations of a variable or function shall have the same type
Rule 6.2.3	Required	The source code used to implement an entity shall appear only once
Rule 6.2.4	Required	A header file shall not contain definitions of functions or objects that are non-inline and have external linkage

Name lookup [basic.lookup]

Rule 6.4.1	Required	A variable declared in an inner scope shall not hide a variable declared in an outer scope

Licensed to: gunjanjain
22 Apr 2025

Rule 6.4.2	Required	Derived classes shall not conceal functions that are inherited from their bases
Rule 6.4.3	Required	A name that is present in a dependent base shall not be resolved by unqualified lookup

Program and linkage [basic.link]

Rule 6.5.1	Advisory	A function or object with external linkage should be introduced in a header file
Rule 6.5.2	Advisory	Internal linkage should be specified appropriately

Storage duration [basic.stc]

Rule 6.7.1	Required	Local variables shall not have static storage duration
Rule 6.7.2	Required	Global variables shall not be used

Object lifetime [basic.life]

Rule 6.8.1	Required	An object shall not be accessed outside of its lifetime
Rule 6.8.2	Mandatory	A function must not return a reference or a pointer to a local variable with automatic storage duration
Rule 6.8.3	Required	An assignment operator shall not assign the address of an object with automatic storage duration to an object with a greater lifetime
Rule 6.8.4	Advisory	Member functions returning references to their object should be ref-qualified appropriately

Types [basic.types]

Rule 6.9.1	Required	The same type aliases shall be used in all declarations of the same entity
Rule 6.9.2	Advisory	The names of the standard signed integer types and standard unsigned integer types should not be used

Standard conversions

The built-in type rules [misra]

Rule 7.0.1	Required	There shall be no conversion from type bool
Rule 7.0.2	Required	There shall be no conversion to type bool
Rule 7.0.3	Required	The numerical value of a character shall not be used
Rule 7.0.4	Required	The operands of bitwise operators and shift operators shall be appropriate
Rule 7.0.5	Required	Integral promotion and the usual arithmetic conversions shall not change the signedness or the type category of an operand
Rule 7.0.6	Required	Assignment between numeric types shall be appropriate

Licensed to: gunjanjain
22 Apr 2025

Pointer conversions [conv.ptr]

Rule 7.11.1	Required	nullptr shall be the only form of the null-pointer-constant
Rule 7.11.2	Required	An array passed as a function argument shall not decay to a pointer
Rule 7.11.3	Required	A conversion from function type to pointer-to-function type shall only occur in appropriate contexts

Expressions

MISRA [misra]

Rule 8.0.1	Advisory	Parentheses should be used to make the meaning of an expression appropriately explicit

Primary expressions [expr.prim]

Rule 8.1.1	Required	A non-transient lambda shall not implicitly capture this
Rule 8.1.2	Advisory	Variables should be captured explicitly in a non-transient lambda

Postfix expressions [expr.post]

Rule 8.2.1	Required	A virtual base class shall only be cast to a derived class by means of dynamic_cast
Rule 8.2.2	Required	C-style casts and functional notation casts shall not be used
Rule 8.2.3	Required	A cast shall not remove any const or volatile qualification from the type accessed via a pointer or by reference
Rule 8.2.4	Required	Casts shall not be performed between a pointer to function and any other type
Rule 8.2.5	Required	reinterpret_cast shall not be used
Rule 8.2.6	Required	An object with integral, enumerated, or pointer to void type shall not be cast to a pointer type
Rule 8.2.7	Advisory	A cast should not convert a pointer type to an integral type
Rule 8.2.8	Required	An object pointer type shall not be cast to an integral type other than std::uintptr_t or std::intptr_t
Rule 8.2.9	Required	The operand to typeid shall not be an expression of polymorphic class type
Rule 8.2.10	Required	Functions shall not call themselves, either directly or indirectly
Rule 8.2.11	Required	An argument passed via ellipsis shall have an appropriate type

Unary expressions [expr.unary]

Rule 8.3.1	Advisory	The built-in unary - operator should not be applied to an expression of unsigned type

Licensed to: gunjanjain
22 Apr 2025

Rule 8.3.2	Advisory	The built-in unary + operator should not be used

Additive operators [expr.add]

Rule 8.7.1	Required	Pointer arithmetic shall not form an invalid pointer
Rule 8.7.2	Required	Subtraction between pointers shall only be applied to pointers that address elements of the same array

Relational operators [expr.rel]

Rule 8.9.1	Required	The built-in relational operators >, >=, < and <= shall not be applied to objects of pointer type, except where they point to elements of the same array

Logical AND operator [expr.log.and]

Rule 8.14.1	Advisory	The right-hand operand of a logical && or || operator should not contain persistent side effects

Assignment and compound assignment [expr.ass]

Rule 8.18.1	Mandatory	An object or subobject must not be copied to an overlapping object
Rule 8.18.2	Advisory	The result of an assignment operator should not be used

Comma operator [expr.comma]

Rule 8.19.1	Advisory	The comma operator should not be used

Constant expressions [expr.const]

Rule 8.20.1	Advisory	An unsigned arithmetic operation with constant operands should not wrap

Statements

Expression statement [stmt.expr]

Rule 9.2.1	Required	An explicit type conversion shall not be an expression statement

Compound statement [stmt.block]

Rule 9.3.1	Required	The body of an iteration-statement or a selection-statement shall be a compound-statement

Selection statements [stmt.select]

Rule 9.4.1	Required	All if ... else if constructs shall be terminated with an else statement
Rule 9.4.2	Required	The structure of a switch statement shall be appropriate

Iteration statements [stmt.iter]

Rule 9.5.1	Advisory	Legacy for statements should be simple
Rule 9.5.2	Required	A for-range-initializer shall contain at most one function call

Licensed to: gunjanjain
22 Apr 2025

Jump statements [stmt.jump]

Rule 9.6.1	Advisory	The goto statement should not be used
Rule 9.6.2	Required	A goto statement shall reference a label in a surrounding block
Rule 9.6.3	Required	The goto statement shall jump to a label declared later in the function body
Rule 9.6.4	Required	A function declared with the [[noreturn]] attribute shall not return
Rule 9.6.5	Required	A function with non-void return type shall return a value on all paths

Declarations

MISRA [misra]

Rule 10.0.1	Advisory	A declaration should not declare more than one variable or member variable

Specifiers [dcl.spec]

Rule 10.1.1	Advisory	The target type of a pointer or lvalue reference parameter should be const-qualified appropriately
Rule 10.1.2	Required	The volatile qualifier shall be used appropriately

Enumeration declarations [dcl.enum]

Rule 10.2.1	Required	An enumeration shall be defined with an explicit underlying type
Rule 10.2.2	Advisory	Unscoped enumerations should not be declared
Rule 10.2.3	Required	The numeric value of an unscoped enumeration with no fixed underlying type shall not be used

Namespaces [basic.namespace]

Rule 10.3.1	Advisory	There should be no unnamed namespaces in header files

The asm declaration [dcl.asm]

Rule 10.4.1	Required	The asm declaration shall not be used

Declarators

Meaning of declarators [dcl.meaning]

Rule 11.3.1	Advisory	Variables of array type should not be declared
Rule 11.3.2	Advisory	The declaration of an object should contain no more than two levels of pointer indirection

Initializers [dcl.init]

Rule 11.6.1	Advisory	All variables should be initialized

Licensed to: gunjanjain
22 Apr 2025

Rule 11.6.2	Mandatory	The value of an object must not be read before it has been set
Rule 11.6.3	Required	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Classes

Class members [class.mem]

Rule 12.2.1	Advisory	Bit-fields should not be declared
Rule 12.2.2	Required	A bit-field shall have an appropriate type
Rule 12.2.3	Required	A named bit-field with signed integer type shall not have a length of one bit

Unions [class.union]

Rule 12.3.1	Required	The union keyword shall not be used

Derived classes

Multiple base classes [class.mi]

Rule 13.1.1	Advisory	Classes should not be inherited virtually
Rule 13.1.2	Required	An accessible base class shall not be both virtual and non-virtual in the same hierarchy

Virtual functions [class.virtual]

Rule 13.3.1	Required	User-declared member functions shall use the virtual, override and final specifiers appropriately
Rule 13.3.2	Required	Parameters in an overriding virtual function shall not specify different default arguments
Rule 13.3.3	Required	The parameters in all declarations or overrides of a function shall either be unnamed or have identical names
Rule 13.3.4	Required	A comparison of a potentially virtual pointer to member function shall only be with nullptr

Member access control

Access specifiers [class.access.spec]

Rule 14.1.1	Advisory	Non-static data members should be either all private or all public

Special member functions

MISRA [misra]

Rule 15.0.1	Required	Special member functions shall be provided appropriately

Licensed to: gunjanjain
22 Apr 2025

Rule 15.0.2	Advisory	User-provided copy and move member functions of a class should have appropriate signatures

Constructors [class.ctor]

Rule 15.1.1	Required	An object's dynamic type shall not be used from within its constructor or destructor
Rule 15.1.2	Advisory	All constructors of a class should explicitly initialize all of its virtual base classes and immediate base classes
Rule 15.1.3	Required	Conversion operators and constructors that are callable with a single argument shall be explicit
Rule 15.1.4	Advisory	All direct, non-static data members of a class should be initialized before the class object is accessible
Rule 15.1.5	Required	A class shall only define an initializer-list constructor when it is the only constructor

Copying and moving class objects [class.copy]

Dir 15.8.1	Required	User-provided copy assignment operators and move assignment operators shall handle self-assignment

Overloading

Overloaded operators [over.oper]

Rule 16.5.1	Required	The logical AND and logical OR operators shall not be overloaded
Rule 16.5.2	Required	The address-of operator shall not be overloaded

Built-in operators [over.built]

Rule 16.6.1	Advisory	Symmetrical operators should only be implemented as non-member functions

Templates

Function template specialization [temp.fct.spec]

Rule 17.8.1	Required	Function templates shall not be explicitly specialized

Exception handling

Throwing an exception [except.throw]

Rule 18.1.1	Required	An exception object shall not have pointer type
Rule 18.1.2	Required	An empty throw shall only occur within the compound-statement of a catch handler

Licensed to: gunjanjain
22 Apr 2025

Handling an exception [except.handle]

Rule 18.3.1	Advisory	There should be at least one exception handler to catch all otherwise unhandled exceptions
Rule 18.3.2	Required	An exception of class type shall be caught by const reference or reference
Rule 18.3.3	Required	Handlers for a function-try-block of a constructor or destructor shall not refer to non-static members from their class or its bases

Exception specifications [except.spec]

Rule 18.4.1	Required	Exception-unfriendly functions shall be noexcept

Special functions [except.special]

Rule 18.5.1	Advisory	A noexcept function should not attempt to propagate an exception to the calling function
Rule 18.5.2	Advisory	Program-terminating functions should not be used

Preprocessing directives

MISRA [misra]

Rule 19.0.1	Required	A line whose first token is # shall be a valid preprocessing directive
Rule 19.0.2	Required	Function-like macros shall not be defined
Rule 19.0.3	Advisory	#include directives should only be preceded by preprocessor directives or comments
Rule 19.0.4	Advisory	#undef should only be used for macros defined previously in the same file

Conditional inclusion [cpp.cond]

Rule 19.1.1	Required	The defined preprocessor operator shall be used appropriately
Rule 19.1.2	Required	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related
Rule 19.1.3	Required	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be defined prior to evaluation

Source file inclusion [cpp.include]

Rule 19.2.1	Required	Precautions shall be taken in order to prevent the contents of a header file being included more than once
Rule 19.2.2	Required	The #include directive shall be followed by either a <filename> or "filename" sequence
Rule 19.2.3	Required	The ' or " or \ characters and the /* or // character sequences shall not occur in a header file name

Licensed to: gunjanjain
22 Apr 2025

Macro replacement [cpp.replace]

Rule 19.3.1	Advisory	The # and ## preprocessor operators should not be used
Rule 19.3.2	Required	A macro parameter immediately following a # operator shall not be immediately followed by a ## operator
Rule 19.3.3	Required	The argument to a mixed-use macro parameter shall not be subject to further expansion
Rule 19.3.4	Required	Parentheses shall be used to ensure macro arguments are expanded appropriately
Rule 19.3.5	Required	Tokens that look like a preprocessing directive shall not occur within a macro argument

Pragma directive [cpp.pragma]

Rule 19.6.1	Advisory	The #pragma directive and the _Pragma operator should not be used

Language support library

Common definitions [support.types]

Rule 21.2.1	Required	The library functions atof, atoi, atol and atoll from <cstdlib> shall not be used
Rule 21.2.2	Required	The string handling functions from <cstring>, <cstdlib>, <cwchar> and <cinttypes> shall not be used
Rule 21.2.3	Required	The library function system from <cstdlib> shall not be used
Rule 21.2.4	Required	The macro offsetof shall not be used

Dynamic memory management [support.dynamic]

Rule 21.6.1	Advisory	Dynamic memory should not be used
Rule 21.6.2	Required	Dynamic memory shall be managed automatically
Rule 21.6.3	Required	Advanced memory management shall not be used
Rule 21.6.4	Required	If a project defines either a sized or unsized version of a global operator delete, then both shall be defined
Rule 21.6.5	Required	A pointer to an incomplete class type shall not be deleted

Other runtime support [support.runtime]

Rule 21.10.1	Required	The features of <cstdarg> shall not be used
Rule 21.10.2	Required	The standard header file <csetjmp> shall not be used
Rule 21.10.3	Required	The facilities provided by the standard header file <csignal> shall not be used

Licensed to: gunjanjain
22 Apr 2025

Diagnostics library

Assertions [assertions]

Rule 22.3.1	Required	The assert macro shall not be used with a constant-expression

Error numbers [errno]

Rule 22.4.1	Required	The literal value zero shall be the only value assigned to errno

General utilities library

Smart pointers [smartptr]

Rule 23.11.1	Advisory	The raw pointer constructors of std::shared_ptr and std::unique_ptr should not be used

Strings library

Null-terminated sequence utilities [c.strings]

Rule 24.5.1	Required	The character handling functions from <cctype> and <cwctype> shall not be used
Rule 24.5.2	Required	The C++ Standard Library functions memcpy, memmove and memcmp from <cstring> shall not be used

Localization library

C library locales [c.locales]

Rule 25.5.1	Required	The setlocale and std::locale::global functions shall not be called
Rule 25.5.2	Mandatory	The pointers returned by the C++ Standard Library functions localeconv, getenv, setlocale or strerror must only be used as if they have pointer to const-qualified type
Rule 25.5.3	Mandatory	The pointer returned by the C++ Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror must not be used following a subsequent call to the same function

Containers library

Sequence containers [sequences]

Rule 26.3.1	Advisory	std::vector should not be specialized with bool

Algorithms library

Algorithms requirements [algorithms.requirements]

Rule 28.3.1	Required	Predicates shall not have persistent side effects

Licensed to: gunjanjain
22 Apr 2025

Mutating sequence operations [alg.modifying.operations]

Rule 28.6.1	Required	The argument to std::move shall be a non-const lvalue
Rule 28.6.2	Required	Forwarding references and std::forward shall be used together
Rule 28.6.3	Required	An object shall not be used while in a potentially moved-from state
Rule 28.6.4	Required	The result of std::remove, std::remove_if, std::unique and empty shall be used

Input/output library

MISRA [misra]

Rule 30.0.1	Required	The C Library input/output functions shall not be used
Rule 30.0.2	Required	Reads and writes on the same file stream shall be separated by a positioning operation

Licensed to: gunjanjain
22 Apr 2025

Appendix B Guideline attributes

Rule	Category	Analysis
Rule 0.0.1	Required	Decidable, Single Translation Unit
Rule 0.0.2	Advisory	Undecidable, System
Rule 0.1.1	Advisory	Undecidable, System
Rule 0.1.2	Required	Decidable, Single Translation Unit
Rule 0.2.1	Advisory	Decidable, Single Translation Unit
Rule 0.2.2	Required	Decidable, Single Translation Unit
Rule 0.2.3	Advisory	Decidable, Single Translation Unit
Rule 0.2.4	Advisory	Decidable, System
Dir 0.3.1	Advisory	
Dir 0.3.2	Required	
Rule 4.1.1	Required	Undecidable, System
Rule 4.1.2	Advisory	Decidable, Single Translation Unit
Rule 4.1.3	Required	Undecidable, System
Rule 4.6.1	Required	Undecidable, System
Rule 5.0.1	Advisory	Decidable, Single Translation Unit
Rule 5.7.1	Required	Decidable, Single Translation Unit
Dir 5.7.2	Advisory	
Rule 5.7.3	Required	Decidable, Single Translation Unit
Rule 5.10.1	Required	Decidable, Single Translation Unit
Rule 5.13.1	Required	Decidable, Single Translation Unit
Rule 5.13.2	Required	Decidable, Single Translation Unit
Rule 5.13.3	Required	Decidable, Single Translation Unit
Rule 5.13.4	Required	Decidable, Single Translation Unit
Rule 5.13.5	Required	Decidable, Single Translation Unit
Rule 5.13.6	Required	Decidable, Single Translation Unit
Rule 5.13.7	Required	Decidable, Single Translation Unit
Rule 6.0.1	Required	Decidable, Single Translation Unit
Rule 6.0.2	Advisory	Decidable, Single Translation Unit
Rule 6.0.3	Advisory	Decidable, Single Translation Unit
Rule 6.0.4	Required	Decidable, Single Translation Unit
Rule 6.2.1	Required	Decidable, System
Rule 6.2.2	Required	Decidable, System
Rule 6.2.3	Required	Decidable, System
Rule 6.2.4	Required	Decidable, Single Translation Unit
Rule 6.4.1	Required	Decidable, Single Translation Unit
Rule 6.4.2	Required	Decidable, Single Translation Unit
Rule 6.4.3	Required	Decidable, Single Translation Unit
Rule 6.5.1	Advisory	Decidable, Single Translation Unit
Rule 6.5.2	Advisory	Decidable, Single Translation Unit
Rule 6.7.1	Required	Decidable, Single Translation Unit
Rule 6.7.2	Required	Decidable, Single Translation Unit

Licensed to: gunjanjain
22 Apr 2025

Rule	Category	Analysis
Rule 6.8.1	Required	Undecidable, System
Rule 6.8.2	Mandatory	Decidable, Single Translation Unit
Rule 6.8.3	Required	Decidable, Single Translation Unit
Rule 6.8.4	Advisory	Decidable, Single Translation Unit
Rule 6.9.1	Required	Decidable, Single Translation Unit
Rule 6.9.2	Advisory	Decidable, Single Translation Unit
Rule 7.0.1	Required	Decidable, Single Translation Unit
Rule 7.0.2	Required	Decidable, Single Translation Unit
Rule 7.0.3	Required	Decidable, Single Translation Unit
Rule 7.0.4	Required	Decidable, Single Translation Unit
Rule 7.0.5	Required	Decidable, Single Translation Unit
Rule 7.0.6	Required	Decidable, Single Translation Unit
Rule 7.11.1	Required	Decidable, Single Translation Unit
Rule 7.11.2	Required	Decidable, Single Translation Unit
Rule 7.11.3	Required	Decidable, Single Translation Unit
Rule 8.0.1	Advisory	Decidable, Single Translation Unit
Rule 8.1.1	Required	Decidable, Single Translation Unit
Rule 8.1.2	Advisory	Decidable, Single Translation Unit
Rule 8.2.1	Required	Decidable, Single Translation Unit
Rule 8.2.2	Required	Decidable, Single Translation Unit
Rule 8.2.3	Required	Decidable, Single Translation Unit
Rule 8.2.4	Required	Decidable, Single Translation Unit
Rule 8.2.5	Required	Decidable, Single Translation Unit
Rule 8.2.6	Required	Decidable, Single Translation Unit
Rule 8.2.7	Advisory	Decidable, Single Translation Unit
Rule 8.2.8	Required	Decidable, Single Translation Unit
Rule 8.2.9	Required	Decidable, Single Translation Unit
Rule 8.2.10	Required	Undecidable, System
Rule 8.2.11	Required	Decidable, Single Translation Unit
Rule 8.3.1	Advisory	Decidable, Single Translation Unit
Rule 8.3.2	Advisory	Decidable, Single Translation Unit
Rule 8.7.1	Required	Undecidable, System
Rule 8.7.2	Required	Undecidable, System
Rule 8.9.1	Required	Undecidable, System
Rule 8.14.1	Advisory	Undecidable, System
Rule 8.18.1	Mandatory	Undecidable, System
Rule 8.18.2	Advisory	Decidable, Single Translation Unit
Rule 8.19.1	Advisory	Decidable, Single Translation Unit
Rule 8.20.1	Advisory	Decidable, Single Translation Unit
Rule 9.2.1	Required	Decidable, Single Translation Unit
Rule 9.3.1	Required	Decidable, Single Translation Unit
Rule 9.4.1	Required	Decidable, Single Translation Unit
Rule 9.4.2	Required	Decidable, Single Translation Unit

Licensed to: gunjanjain
22 Apr 2025

Rule	Category	Analysis
Rule 9.5.1	Advisory	Decidable, Single Translation Unit
Rule 9.5.2	Required	Decidable, Single Translation Unit
Rule 9.6.1	Advisory	Decidable, Single Translation Unit
Rule 9.6.2	Required	Decidable, Single Translation Unit
Rule 9.6.3	Required	Decidable, Single Translation Unit
Rule 9.6.4	Required	Undecidable, System
Rule 9.6.5	Required	Decidable, Single Translation Unit
Rule 10.0.1	Advisory	Decidable, Single Translation Unit
Rule 10.1.1	Advisory	Decidable, Single Translation Unit
Rule 10.1.2	Required	Decidable, Single Translation Unit
Rule 10.2.1	Required	Decidable, Single Translation Unit
Rule 10.2.2	Advisory	Decidable, Single Translation Unit
Rule 10.2.3	Required	Decidable, Single Translation Unit
Rule 10.3.1	Advisory	Decidable, Single Translation Unit
Rule 10.4.1	Required	Decidable, Single Translation Unit
Rule 11.3.1	Advisory	Decidable, Single Translation Unit
Rule 11.3.2	Advisory	Decidable, Single Translation Unit
Rule 11.6.1	Advisory	Decidable, Single Translation Unit
Rule 11.6.2	Mandatory	Undecidable, System
Rule 11.6.3	Required	Decidable, Single Translation Unit
Rule 12.2.1	Advisory	Decidable, Single Translation Unit
Rule 12.2.2	Required	Decidable, Single Translation Unit
Rule 12.2.3	Required	Decidable, Single Translation Unit
Rule 12.3.1	Required	Decidable, Single Translation Unit
Rule 13.1.1	Advisory	Decidable, Single Translation Unit
Rule 13.1.2	Required	Decidable, Single Translation Unit
Rule 13.3.1	Required	Decidable, Single Translation Unit
Rule 13.3.2	Required	Decidable, Single Translation Unit
Rule 13.3.3	Required	Decidable, System
Rule 13.3.4	Required	Decidable, Single Translation Unit
Rule 14.1.1	Advisory	Decidable, Single Translation Unit
Rule 15.0.1	Required	Decidable, Single Translation Unit
Rule 15.0.2	Advisory	Decidable, Single Translation Unit
Rule 15.1.1	Required	Undecidable, System
Rule 15.1.2	Advisory	Decidable, Single Translation Unit
Rule 15.1.3	Required	Decidable, Single Translation Unit
Rule 15.1.4	Advisory	Decidable, Single Translation Unit
Rule 15.1.5	Required	Decidable, Single Translation Unit
Dir 15.8.1	Required	
Rule 16.5.1	Required	Decidable, Single Translation Unit
Rule 16.5.2	Required	Decidable, Single Translation Unit
Rule 16.6.1	Advisory	Decidable, Single Translation Unit
Rule 17.8.1	Required	Decidable, Single Translation Unit

Licensed to: gunjanjain
22 Apr 2025

Rule	Category	Analysis
Rule 18.1.1	Required	Decidable, Single Translation Unit
Rule 18.1.2	Required	Decidable, Single Translation Unit
Rule 18.3.1	Advisory	Decidable, Single Translation Unit
Rule 18.3.2	Required	Decidable, Single Translation Unit
Rule 18.3.3	Required	Decidable, Single Translation Unit
Rule 18.4.1	Required	Decidable, Single Translation Unit
Rule 18.5.1	Advisory	Undecidable, System
Rule 18.5.2	Advisory	Decidable, Single Translation Unit
Rule 19.0.1	Required	Decidable, Single Translation Unit
Rule 19.0.2	Required	Decidable, Single Translation Unit
Rule 19.0.3	Advisory	Decidable, Single Translation Unit
Rule 19.0.4	Advisory	Decidable, Single Translation Unit
Rule 19.1.1	Required	Decidable, Single Translation Unit
Rule 19.1.2	Required	Decidable, Single Translation Unit
Rule 19.1.3	Required	Decidable, Single Translation Unit
Rule 19.2.1	Required	Decidable, Single Translation Unit
Rule 19.2.2	Required	Decidable, Single Translation Unit
Rule 19.2.3	Required	Decidable, Single Translation Unit
Rule 19.3.1	Advisory	Decidable, Single Translation Unit
Rule 19.3.2	Required	Decidable, Single Translation Unit
Rule 19.3.3	Required	Decidable, Single Translation Unit
Rule 19.3.4	Required	Decidable, Single Translation Unit
Rule 19.3.5	Required	Decidable, Single Translation Unit
Rule 19.6.1	Advisory	Decidable, Single Translation Unit
Rule 21.2.1	Required	Decidable, Single Translation Unit
Rule 21.2.2	Required	Decidable, Single Translation Unit
Rule 21.2.3	Required	Decidable, Single Translation Unit
Rule 21.2.4	Required	Decidable, Single Translation Unit
Rule 21.6.1	Advisory	Undecidable, Single Translation Unit
Rule 21.6.2	Required	Decidable, Single Translation Unit
Rule 21.6.3	Required	Decidable, Single Translation Unit
Rule 21.6.4	Required	Decidable, System
Rule 21.6.5	Required	Decidable, Single Translation Unit
Rule 21.10.1	Required	Decidable, Single Translation Unit
Rule 21.10.2	Required	Decidable, Single Translation Unit
Rule 21.10.3	Required	Decidable, Single Translation Unit
Rule 22.3.1	Required	Decidable, Single Translation Unit
Rule 22.4.1	Required	Decidable, Single Translation Unit
Rule 23.11.1	Advisory	Decidable, Single Translation Unit
Rule 24.5.1	Required	Decidable, Single Translation Unit
Rule 24.5.2	Required	Decidable, Single Translation Unit
Rule 25.5.1	Required	Decidable, Single Translation Unit
Rule 25.5.2	Mandatory	Decidable, Single Translation Unit

Licensed to: gunjanjain
22 Apr 2025

Rule	Category	Analysis
Rule 25.5.3	Mandatory	Undecidable, System
Rule 26.3.1	Advisory	Decidable, Single Translation Unit
Rule 28.3.1	Required	Undecidable, System
Rule 28.6.1	Required	Decidable, Single Translation Unit
Rule 28.6.2	Required	Decidable, Single Translation Unit
Rule 28.6.3	Required	Decidable, Single Translation Unit
Rule 28.6.4	Required	Decidable, Single Translation Unit
Rule 30.0.1	Required	Decidable, Single Translation Unit
Rule 30.0.2	Required	Undecidable, System

Licensed to: gunjanjain
22 Apr 2025

Appendix C Glossary

Callback

A *callback* is a function that is called indirectly via a function pointer or a *handle*.

Dataflow anomaly

The state of a variable at a point in a program can be described using the following terms:

- Undefined (U) — the value of the variable is indeterminate; and
- Referenced (R) — the variable is used in some way (e.g. in an expression); and
- Defined (D) — the variable is explicitly initialized or assigned a value.

Given the above, the following *dataflow anomalies* can be defined:

- UR dataflow anomaly — variable not assigned a value before the specified use; and
- DU dataflow anomaly — variable is assigned a value that is never subsequently used; and
- DD dataflow anomaly — variable is assigned a value twice with no intermediate use.

DD dataflow anomaly

See *dataflow anomaly*.

Declaration

A *declaration* introduces the name of an *entity* into a *translation unit* (see [basic.def]/1).

An *entity* may be *declared* several times. The first *declaration* of an *entity* in a *translation unit* is called an *introduction*. All subsequent *declarations* are called *redeclarations*.

A *definition* is a *declaration*, as described in [basic.def]/2.

Definition

See *declaration*.

Directly enclose

See *enclose*.

DU dataflow anomaly

See *dataflow anomaly*.

Enclose

A statement S1 *directly encloses* a statement S2 if:

- S1 is a *labeled-statement*, and S2 is the contained statement; or
- S1 is a *compound-statement*, and S2 is any statement of its *statement-seq*; or
- S1 is a *selection-statement*, and S2 is any of its statements (but not its *init-statement*); or
- S1 is an *iteration-statement*, and S2 is the contained statement (but not an *init-statement*).

A statement S1 encloses a statement S2 if:

- S1 *directly encloses* S2; or
- S1 *directly encloses* a statement S3 and S3 *encloses* S2.

Licensed to: gunjanjain
22 Apr 2025

General manager

A *manager class*, as defined in Rule 15.0.1.

Header file

A *header file* is considered to be any file that is included during preprocessing (for example via the `#include` directive), regardless of its name or suffix.

Infeasible path

Infeasible paths occur where there is a syntactic path to a code fragment, but the semantics ensure that the control flow path will not be executed. For example:

```
if ( u32 < 0 )
{
  // An unsigned value will never be negative,
  // so code in this block will never be executed.
}
```

Introduction

See *declaration*.

Manager class

A class that is either a *scoped manager*, a *unique manager*, or a *general manager* as defined in Rule 15.0.1.

NDR

NDR is an abbreviation for *no diagnostic required*.

ODR

ODR is an abbreviation for the *one-definition rule*.

Persistent side effect

A *side effect* is said to be *persistent* at a particular point in execution if it might have an effect on the execution state at that point. All of the following *side effects* are *persistent* at a given point in the program:

- Modifying a file, stream, etc.;
- Modifying an object, including via a pointer or reference;
- Accessing a *volatile* object;
- Raising an exception that transfers control outside of the current function.

When a function is called, it may have *side effects*. Modifying a file or accessing a *volatile* object are persistent as viewed by the calling function. However any objects modified by the called function, whose lifetimes have ended by the time it returns, do not affect the caller's execution state. Any *side effects* arising from modifying such objects are not *persistent* from the point of view of the caller.

The determination of whether a function has *persistent side effects* takes no consideration of the possible values for parameters or other non-local objects.

Polymorphic class

A polymorphic class is a class that declares or inherits a virtual function.

Licensed to: gunjanjain
22 Apr 2025

RAII

RAII is an abbreviation for Resource Acquisition Is Initialization, which is a programming idiom for scope-based resource management that binds the ownership of a resource to the lifetime of an object — the resource is acquired during construction and released during destruction.

Redeclaration

See *declaration*.

Scoped manager

A *manager class*, as defined in Rule 15.0.1.

Transient lambda

A lambda is *transient* when:

- It is immediately invoked; or
- It is passed to a function that does not *store* it.

A function does not *store* a lambda when:

- The function is defined in the same *translation unit* as the lambda; and
- The lambda is only copied or moved when it is passed as an argument; and
- The function only calls the lambda and/or passes the lambda to another function that does not *store* it.

UR dataflow anomaly

See *dataflow anomaly*.

Unique manager

A *manager class*, as defined in Rule 15.0.1.

Use / used / using

An object is *used* if:

- It is the subject of a cast; or
- It is explicitly initialized at declaration time; or
- It is an operand in an expression; or
- It is referenced.

A function is *used* as defined in Rule 0.2.4.

A type is *used* as defined in Rule 0.2.3.

Unscoped enumeration type

A type created with the `enum` keyword that is not created as `enum class` or `enum struct`. Values of such a type will be subject to *integral promotion*.

Unused

An entity is *unused* if it is not *used*.

Licensed to: gunjanjain
22 Apr 2025

ISBN 978-1-911700-10-4 paperback
ISBN 978-1-911700-11-1 PDF

Licensed to: gunjanjain
22 Apr 2025